Perl 语言入门

（第四版）

作者：**Brian d foy, Tom Phoenix, Randal L.Schartz**

译者： **lebk** 校对： 鄢 元 满

目 录

前　言：

感谢我的家人。是你们给与了我一切。无论走到哪里，你们都让我感到温暖和充满勇气。

感谢我的女朋友：红梅。你对我的照顾和给予我的爱，让我的生命变得完整。

感谢我的朋友阿满在百忙之中抽出时间校对。但本书质量应完全由译者负责。

感谢出现在我生命中所有的人。那些和我一起逃学，一起罚站，一起偷邻居甘蔗的伙伴；那些和我一起抽烟，一起喝酒，一起唱歌侃大山的同学；那些天冷提醒我穿衣，过马路提醒我小心的知己朋友；还有那些和我斗嘴的，和我打架的，是你们锻炼了我的口才，是你们强壮了我的身体。你们还好吗？

最后我想说：当你很饿，面对美味佳肴，请不要急于动筷子；当你很渴，面对清泉甘露，请不要急于动嘴；当你很冷，面对暖炕热被，请不要急于动腿。当你所渴望的东西正在你眼前的时候，请放慢你的脚步，用心去感受。

lebk

第一章 概 述

欢迎使用小骆驼书。

本书是第四版，自 93 年以来，有超过 50 万的读者喜欢它。至少，我们希望他们喜欢它。不管怎样，我们写此书时非常开心◆。

◆本书第一版由 Randal L.Schwartz 著，第二版由 Randal，Tom Christiansen 著，第三版由 Randal，Tom Phoenix 著, 本版由 Randal, Tom Ponenix, 和 brian foy 著。因此，在本版中，当说“我们”时，指的是最后三位。现在，你可能猜想，为什么在第一页就说写本书我们非常开心（过去时态），理由很简单：因为我们是从后往前写的。这听起来很奇怪。但是，坦白讲，当写完索引后，剩下的就变的很容易了。

1．1 问题和解答

你或许对 Perl 有些疑问，也可能是针对本书，特别是当你已经大致浏览本书后。因此，我们将用本章来回答这些问题。

1．1．1 本书适合你吗？

如果你和我们类似，那你很可能正站在书架前◆，考虑是否要买这本羊骆驼书来学习 Perl 或是买另一本由蛇（有迂回的含义），饮料，或者一些字母命名的语言◆的书。你站了两分钟，书店经理走过来通知你这不是图书馆◆，你要么买要么快点离开。可能，你利用这两分钟时间来查看一个 Perl 程序，来了解其强大功能以及它能完成怎样的工作。如果是那样的话，您因该浏览本章剩下的章节。

◆实际上，如果和我们一样，你因该站在图书馆，而不是书店。当然我们有一点吝啬。

◆在你写信告诉我们那是段愉快，而非迂回（伤脑筋）的历程之前，其实我们想的是 CORBA。

◆除非它是，否则

1．1．2 为什么如此多的脚注？

感谢你注意到这些。本书中有大量的脚注。忽略它们就行了。我们需要脚注的原因是 Perl 中有大量的异常。这是好事情，因为生活本身就充满了意外。

但这并不意味着我们不能老实的写：“The fizzbin operator frobnicates the hoozistatic variables”◆，而不写任何脚注来描述异常情况。事实上，我们相当老实，所以我们写了脚注。但你可以老老实实的忽略它们。(这种解决办法听起来相当有趣)。许多异常和移植性相关。Perl 最早在 Unix 系统中使用，因此它和 Unix 渊源极深。但只要有可能，我们就演示它非预期的

行为，无论这种结果是由 non-Unix 系统，还是别的原因引起的。我们希望那些对 Unix 一无所知的读者也认为本书是一本好的 Perl 教程。（同时，也能附带学些 Unix 的知识）。

其余的大量的异常都和“80/20”规则相关。我们是指 80% 的 Perl 可用 20%的篇幅介绍。剩下的 20%需要 80%篇幅。为了保持本书的大小，我们将在正文中介绍最常用的，最容易讲明白的内容，在脚注中介绍别的（它们字体更小，因此可以在相同的空间内写入给多的内容）◆但你学习本书时，如果没有阅读脚注，很可能在后面的章节中需要回过头来查看。如果是那样，或者在学习的过程中，出于好奇，那就读读这些脚注吧。尽管大多数脚注只是一些计算机的笑话。

◆我们甚至讨论为了节省纸张，把整本书用脚注组成，但脚注跟着脚注看起来有些疯狂。

1．1．3 练习题和解答呢？

练习题在每章的末尾。由于我们三人向几千人介绍相同的材料◆。因此我们仔细的选择了这些习题，让你更有机会犯错。

◆不是同时的

并不是我们希望你犯错，而是你需要这种经验。因为在 Perl 生涯中，这些是你最可能犯的错误，因此我们现在就应当提醒你。那些在阅读本书时犯的错误，可能不会在项目的终止日期时重犯。当你出错是我们会帮助你找到错误的原因；附录 A 有每一个习题的解答，和小段注释。当完成习题时请检查答案。

不要在仔细尝试之前看答案。自己解决比读解答学的更牢靠。如果没有找到答案，也不要太伤心。继续阅读下一章。如果你完全没有犯错误，也因当在完成时看下答案。解答的注释中可能有关于这段程序的一些不明显地方的介绍。

1．1．4 习题前的数字是什么意思？

每一习题前都有一个由中括号括起来的数字，像下面这样：

[2]习题前面方括号中的数字 2 是什么含义？

这个数字是我们关于你大概要花多长时间来解决这道题的（非常粗略的）估计。由于非常粗略，当你在一半或两倍的时间内完成（书写，调试，测试）时，也不必太惊讶。另外，当被题目难住时，我们也不会告诉任何人你偷看了附录 A。

1．1．5 如果我是 Perl 教师，怎么办呢？

如果你是 Perl 的教师，决定采用本书作为教科书（过去的年月里，许多人采用了它），你因当知道我们尽力使每一组联系能让大多数学生在 45 分钟～1 小时内完成，其间有些休息时间。一些练习可能很快就完成了，另一些也许花的时间要长些。那是因为，当我们写下了方括号中的数字时，我们发现不知道怎么把它们加起来。（幸运的是，我们知道怎样用计算机来做到）。

1．2　Perl 代表什么？

Perl 一般被称为“实用报表提取语言”（Practical Extraction and Report Language），虽然有时被称做“病态折中垃圾列表器”(Pathologically Eclectic Rubbish Lister)。它是术语，而不仅仅是简写，Perl 的创造者，Larry Wall 提出第一个，但很快又扩展到第二个。那就是为什么“Perl”没有所有字母都大写。没必要争论那一个正确，Larry 两个都认可。

你也可能看到“perl”，所有的字母都是小写的。一般，“Perl”，有大写的 P，是指语言本身，而“perl”，小写的 p，是指程序运行的解释器。

1．2．1　Larry 为什么发明 Perl？

Larry 在 80 年代中期发明了 Perl 语言，当时他想从像新闻组邮件那样的文件中产生一些有用的报表给一个 bug 报告系统，awk 语言不能胜任这任务。Larry，作为一个懒惰的程序员◆，为了彻底的解决这个问题，决定发明一种一般用途的工具，至少还能在一个不同的地方使用。这次努力的结果就是 Perl V0。

◆我们不是无理的说 Larry 很懒惰；实际上懒惰是一种美德。手推车是由那些懒于搬东西的人发明的；文字是由那么懒于记忆的人发明的。Perl 是由那些不创造一种新的语言就懒于完成任务的人发明的。

1．2．2 为什么 Larry 不采用别的语言？

程序语言本身没有缺陷，有吗？但是，在 Larry 那个时候，他没有找到满足需要的语言。如果现在的某种语言在那个时代就已产生，很可能 Larry 就会采用它。他需要能像 shell 或 awk 那样快速编码，同时具有如 grep, cut, sort, sed◆等这些高级工具的强大功能，但又不采用像 C 那样的语言。

◆如果不知道它们，不用担心。这些是 Larry 的 Unix 工具盒中的程序。但它们还不能解决手头的工作。

Perl 填补了低级语言（如 C，C++，汇编语言）和高级语言（如 shell 编程）的空白。低级语言通常难于编码，并且丑陋，但速度快，且无限制；高级语言，在速度上，很难超过书写良好的低级语言。在低级语言里，你几乎能完成任何事。高级语言，正好相反，一般速度慢，困难，丑陋，有限制；如果没有系统提供的函数，shell，批处理语言能完成的工作相当有限。Perl 简单，几乎是无限制的，速度快，也有些丑陋。

让我们从另一个角度来看关于 Perl 的这四点：

第一，Perl 简单。如你将要见到的，这意味着容易使用。但不是特别容易学习。如果学习开车，你花数周或数月学习，然后就很容易的开车了。当你花了许多时间来学习 Perl 时，Perl 对你来说就简单了◆。

◆当然，我们并不希望你翻车(☺)。

Perl 几乎没有限制。几乎没有什么事不能由 Perl 来完成。你一般不希望用 Perl 来书写内核级的中断驱动程序（虽然 Perl 能完成）。但针对一般工作中遇到的问题，从一次性程序到工业级的运用，Perl 都能出色的完成。

Perl 速度快。那是由于，所有的 Perl 开发者都使用 Perl，他们希望它快。如果某人想加一个很酷的功能到 Perl 中，但它会降低其它程序的速度，Larry 基本上会拒绝添加它除非找到一个方法使它足够快。

Perl 有些丑。这是事实。O 'Reilly 给 Perl 的图标是骆驼，这种动物是著名的骆驼书（也被称为 Perl 语言编程）的封面，还有它的兄弟-本书（它的姐妹，羊驼书（Alpaca））。骆驼有些丑。但是它们努力工作，即便在艰苦的环境中。无论什么困难骆驼都能完成任务，虽然他们不好看，不好闻，有时还向你吐唾沫。Perl 有些像它。

1．2．3 Perl 容易学习吗？

Perl 容易使用，但有些难学。当然，这具有普适性。设计 Perl 时，Larry 做了学多权衡。当遇到能让程序员更容易使用，但对于初学者难于学习时，Larry 通常倾向于前一种。那是因为，你只学习一次，而将重复使用◆。Perl 有学多做法来节约程序员的时间。例如学多函数都有默认值；通常，这些默认行为就是你需要的。因此，你将节约学多时间来写像下面这样的代码◆：

◆如果你每周或每月只花几分钟来使用某种语言。你可能选择那些容易学习的语言，因为在下次使用之前你不需要记住它们。Perl 是给那些每天至少花 20 分钟来写程序的人。

◆我们将不详细解释。本例是将文件中某种格式的数据，换成另一种。所有功能都能在本书中均有找到答案。

```
while(<>){
      chomp;
      print join(“\n”,(split /:/))[0,2,1,5]),”\n”;
}
```

如果不利用 Perl 的默认值和简写，本段代码大约会长 10~12 倍，这将花更多时间来阅读和书写。并且由于有更多的变量，将难于维护和调试。如果你懂一点 Perl，没有看见代码中的变量，那只是部分问题。它们都使用的默认值。为了减轻程序员的负担，不得不增加学习的代价，因此你因当学习这些默认值和简写。

一个很好的类比，是英语中的单词。例如，“will not”和“won’t”含义相同。但大多数人说“won’t”而非“will not”，因为这将节约时间，并且每个人都知道它们有相同含义。同样的，Perl 也把一些常用的语句以一种更简略的形式来表达，就像语言那样更快的“说”出来，并且被同行所理解。

一旦熟悉了 Perl，将发现比 shell 引用（或 C 声明）花更少的时间，你将有更多的时间在网上冲浪，因为 Perl 的强大能力。Perl 设计成能让你仅用数行就能漂亮的解决问题。你可以把这些工具带到下一份工作中，因为 Perl 具有很高的移植性，因此你将有更多的时间冲浪。

Perl 是高级语言。这意味着，代码很紧凑，通常 Perl 程序大约是它对应的的 C 程序的 1/4 到 3/4 长。这使得 Perl 程序的读，写，调试，维护速度都更快。当整个程序在一屏中，不需要向上向下滚动查看时，编程将更容易。并且，由于程序中 bugs 的数量大致和它的长度成正比◆（而非和程序的函数），这就意味着，平均起来，短一些的 Perl 程序意味着更少的 bugs。

◆当程序超过一屏时，bugs 数量会突增。

1．2．4 Perl 为什么如此流行?

Larry 使用了 Perl 一段时间，做了些修补后，把它发给新闻组上的用户了，通常被称为“网上”。成千上万的用户问他，做这个那个以及别的许多问题的方法，Larry 从没有预想过他的小 Perl 能处理那些问题。

结果，Perl 持续增长。它的特性越来越多。其移植性越来越强。曾经的一门小语言成长为拥有上千页在线文档，数十本书，几个主流新闻组（还有大量非主流新闻组），每一天几乎每个当代系统上都有无数的读者和实现者。当然，不要忘了这本小骆驼书。

1．2．5 Perl 正在发生怎样的事情?

Larry 最近已经不写代码了，但他仍然领导着 Perl 的发展，并作出关键决定。Perl 主要由一群勤劳的被称做 Perl 5 守门人（Perl 5 Porters）维护。你可以在 perl5-porters@perl.org 中查看他们的工作进展和讨论。

当我们写做此书时(2005 年 3 月),Perl 已经发生了许多变化。在过去的几年中，许多人都忙于下一个主要版本：Perl 6。不要把你的 Perl 5 扔到一旁，因为它仍是当前最新和最稳定的版本。我们并不期望 Perl 6 的稳定版本会很快出现。当 Perl 6 出现时，Perl 5 也不会消失，几年之内，许多人将同时使用这两个版本。Perl 5 的维护者经常把 Perl 6 中的好点子加到了 Perl 5 中。

2000 年，Larry Wall 第一次提出了下一个主要版本是 Perl 社区对 Perl 的重写。自那以后，一个新的解释器“Parrot”就进入人们的视野，但并没影响到普通用户。今年（2005 年），唐宗汉（Autrijus Tang）开始利用 Haskell 实现作为轻量级的 Perl 6 的 Pugs（Perl User Golfing System）。来自全球的 Perl 和 Haskell 开发者提供了帮助。在 http://dev.perl.org/perl6 和 http://www.pugscode.org/ 中可以了解到更多信息。

1．2．6 Perl 擅长什么?

Perl 擅长写那些需要在短时间内完成的程序。对于那些需要数十个程序员，花费数年的程序，Perl 也能很好的胜任。当然，更多的情况是你将写那些从开始构思到实际测试代码只需几十分钟的程序。

Perl 被设计为：90%处理文本，10%针对其它情况。这种能力基本上能满足当今的编程任务。在理想情况下，每一个程序员懂得每一种语言；对于不同的项目将采用最合适的语言。大多数情况，你要选择 Perl◆。当 Larry 发明 Perl 的时候，Tim Berners-Lee 还没有 web 的丝毫想法，但它们是互联上的完美联姻。许多人声称 90 年代初 Perl 的发展使得内容能快速转换为 HTML 格式在网上传输，而没有内容 Web 是不存在的。当然，Perl 是一种优秀的书写 CGI 脚本（由 web 服务器运行的程序）的语言，因此许多人如今仍说：“CGI 仅是 Perl 吗？”或者“为什么不说 Perl 而说 CGI？”，这些论述很有意思。

◆不要较真。如果想知道 Perl 是否比 X 语言好，那同时学习它们，看那种语言用的最多。这对你是最好的方法。并且，同时学习两种语言，你将更好的理解它们。花这些时间是值得的。

1．2．7 Perl 不擅长什么？

Perl 擅长许多事，那么什么是它不擅长的呢？不应当使用 Perl 来产生二进制码。那些程序可以给别人，或卖给别人，而他们不能看到程序内部的秘密，同时也不能维护和调试代码。当把 Perl 程序给别人时，通常给他们的是源代码而非二进制程序。

当想要二进制程序时，我们没告诉你不可能。如果人们能安装和运行你的程序，它们也能反编译出来，无论是哪种语言。当然，这可能和你最初的源代码不同，但它们在某种程度上类似。要保护你的程序，最好的方法是，找些律师，写一份 license：“你可以利用代码做这个，不能做那个。如果违反这个规则，那我们将有律师找你的麻烦，保证让你后悔”。

1．3 怎样获得 Perl？

你可能已经有了。至少，我们去的地方都装有它。它被移植到许多系统中，系统管理员通常把它安装在每一台机器上。即便如此，如果系统上还没有安装，你也可以免费获得。

Perl 通过两种许可证（license）发布。对于大多数 Perl 用户而言，任意许可证都足够了。如果想修改 Perl，那你因当仔细的阅读这些许可证，这里面有对修改代码的一些小的限制。对于不修改 Perl 的用户来讲，“它是完全免费的。”

因此，它是免费的，可以在任何具有 C 编译器的机器上运行。下载它，输入几个命令，它就能自动配置和安装。甚至，你可以让系统管理员帮你完成这些事◆。除了 Unix 和类 Unix 系统之外，人们特别喜欢将它移植到其他系统中，例如 Macintosh◆，VMS, OS/2, MS/DOS, Windows，当你阅读本书时，它可能已经被移植到更多的系统中了◆。在这些新系统中安装 Perl，一般要比在 Unix 中安装容易。请查看“CPAN”中“移植”这一部分以了解更多信息。

◆如果系统管理员不能安装 Perl，那雇用他有什么用呢？如果你说服他们有困难，去买份 pizza，没有几人能对一份免费的 pizza 说不。

◆Mac 操作系统中使用的是 MacPerl。如果你有 Mac 操作系统，它是类 Unix 系统，那么你拥有主流的 Perl。

◆现在还不适合掌上机，因为它太大了。曾有谣言说它能在 WinCE 上运行。

1．3．1 什么 CPAN？

CPAN 是全面 Perl 归档网络（Comprehensive Perl Archive Network）的缩写，那是一个值得常去的地方。这里有 Perl 源码，容易安装到非类 Unix 系统的 Perl◆，例子，文档，Perl 扩展部分，Perl 归档信息等。简言之，CPAN 是全面的。

◆一般在 Unix 系统中，最好是自己编译 Perl。在别的没有 C 编译器或其它工具的系统中，CPAN 上提高了二进制的 Perl。

CPAN 在全世界有上百个镜像站点。在 http://serach.cpan.org/ 和 http://kobesearch.cpan.org 上可以找到他们。如果不能上网，你可以从 CD-ROM 或 DVD-ROM 上得到 CPAN 中的部分内容。去附近的技术书店，找一份最近的 CPAN 归档光盘，因为 CPAN 每天都在更新。2 年以前的文档已经是古董了。最好找一个能上网的朋友，让它帮你烧录份最新的 CPAN 文档。

1. 3. 2 怎样获得支持?

你有所有的源码，所以得自己修复错误（bugs）。

听起来不太好，是吧？但是好事情。由于没有对 Perl 源码的限制，任何人都可以修改源码来解决 bugs。事实上，当你发现并确认一个 bug 时，也许已经有人解决它了。全世界有几千人在维护 Perl。

我们并不是说 Perl 有许多 bugs，但它是程序，每一个程序都至少有一个 bug。为了说明为什么拥有源码会如此有用，想象你有另一种由一个强大组织，其拥有者是一个光头的亿万富翁所提供的名为 Forehead 的语言，(这仅是一个假设。事实上，我们知道没有一种语言叫做 Forehead)。现在想想当你发现 Forehead 中有 bug 时，你能做什么。首先，你报告它。其次，你期待他们解决它，希望他们尽快解决它，并且下一版要价不要太高。在下一版中，当增加新功能后，不要引入新的 bugs，希望那个公司不会违反反托拉斯法而导致破产。

对于 Perl，你有源码。如果不幸找到的 bug 没人解决，这基本上是不可能的，你可以雇佣一个和十个程序员来解决它。同理，如果买了一台新机器，Perl 不能在上面运行，你可以自己移植它。现在，当你需要一个不存在的功能时，你知道该怎么做了。

1. 3. 3 有其它的支持吗?

当然。我们最喜欢的是 Perl Mongers。这是由全球的 Perl 用户组组成的。在 http://www.pm.org/ 可以找到更多的信息。可能你附近的用户组中就有一个专家，或者有人认识某个专家。如果没有这样的组，你可以创建一个。

当然，为了获得帮助，首先，不应该忽略文档。除了 用户手册外◆，也可以从 CPAN：http://www.cpan.org 上找到合适的文档，当然还有其它站点，如 http://perldoc.perl.org，上面有 Perl 文档的 HTML 和 PDF 版本，http://www.perldoc.org 可以让你搜寻不同版本的文档，http://faq.perl.org 上有最新版本 perlfaq.

◆用户手册 s 是类 UNIX 系统上的一种文档，如果没有 Unix 系统，Perl 有一份针对你的文件系统的文档。

另一份权威的资料是 O'Reilly 出版的 Perl 编程语言（Programming Perl）, 通常被称做“骆驼书”，因为封面上有骆驼。(本书被成为“小骆驼书”)。骆驼书中有详尽的参考信息，一些教程，和大量的关于 Perl 的额外信息。还有一本由 Johan Vromans（O'Reilly）编写的 Perl 5 的袖珍书，你可以随时把它带在身边（或放在口袋里）。

如果想问问题，网上有大量的新闻组和邮件列表◆。每一天的任意时刻，都有某个时区的 Perl 专家在新闻组中回答问题；在 Perl 帝国里太阳永远都不西沉的。这意味着，你问一个问题，通常数几分钟内就有答案。如果不先查看 FAQ，很快就有人热情的帮助你。

◆许多邮件列表可以在 http://lists.perl.org 中得到。

官方的 Perl 新闻组在 comp.lan.perl.*的某些层级上. 当写作本书时，有五个，当它们经常改变。你（或者某个别的负责你们那里 Perl 的人）应当订阅 comp. lang. perl. anounce, 上面有 Perl 的重要通知，特别是关于安全方面的通知。如果需要如何使用新闻组的帮助，请教附近的专家。

有几个著名的讨论 Perl 的 Web 站点。其中最著名的是:Perl Monastery (http://www.perlmonks.org), 其中有许多 Perl 书籍的作者和专栏作家，至少包括本书的两位作者。你可以查看 http://learn.perl.org/ 和它相关的邮件列表 beginners@perl.org.

如果需要付费的帮助，有一些公司乐意为你提供任意数量的收费服务的帮助。当然也有许多免费的服务。

1．3．4 当发现 Perl 中有错误时，该怎么办?

当发现 bug 时，第一件事是再次◆检查文档◆。Perl 有许多特殊性质，和不符规则的地方，你应当确认你发现的是某个特殊性质还是 bug。检查你的 Perl 是否是老版本；也许你发现的问题在新一些的版本中已经得到了解决。

◆甚至 Larry 也承认，他经常参考这些文档。

◆甚至两次或三次。大多数时候，当我们查看文档以得到找到某一异常行为的解释是，通常得到某些别的细微差别的介绍。

当几乎认定发现的是一个真正的 bug 时，问问你周围的人。问问那些工作的，或者附近的 Perl 协会中的人。通常，你发现的仍是一个特殊性质而非 bug。

当你肯定发现的是 bug，那么准备一个测试案例(test case)。（你以前没有做过吗？）理想的测试案例是，一段小的的程序，任何 Perl 用户都能执行它，并且能得到和你一致的结果。准备好一个能反映这个 bug 的测试案例后，应当用 perlbug（Perl 中带有它)这个工具来报告它。它会把这个问题用邮件发给 Perl 的开发者，因此在准备好测试案例之前不要随便使用 perlbug。

当把 bug 报告出去后，你的事情就完成了，通常能在几分钟内得到回应。一般，你会得到一块小的补丁，然后你的 Perl 就能恢复正常。当然，你也可能得不到任何回答，因为 Perl 的开发者没有义务来阅读这些 bug 报告。但我们都热爱 Perl，我们不希望 Perl 中有任何错误从我们的眼皮底下溜走。

1．4 怎样才能写一个 Perl 程序？

是时候问这个问题了（也许你还没有呢）。Perl 程序是文本类型的；可以用你最喜欢的文本编辑器来创建它们（你并不需要任何特别的开发环境，虽然有一些商业公司提供。我们对于这些工具都使用不多，所以不够资格推荐它们。）

应当使用程序员的文本编辑器(programmer’s text editor)，而不是普通的编辑器。它们有什么不同点呢？一般，程序员的编辑器能提供一个程序员所需要的功能，例如缩进，或非缩进一块代码，能匹配对应的花括号等。在 Unix 系统中两个最流行的程序员编辑器是 emacs 和 vi（以及它们的克隆和变种）。BBEdit 和 Alpha 是 Mac 系统中两个优秀的编辑器。在 Windows 平台上，口碑很好的编辑器是 UltraEdit 和 PFE(程序员喜欢的编辑器(Programmer’s Favorite Editor,))。Perlfaq2 上列有几个其它的编辑器。询问你当地的专家，让他推荐你机器上的编辑器。

对于本书的练习题而言，其代码长度都在 20 或 30 行之内，任意编辑器都能胜任。

少数的初学者使用字处理软件而非文本编辑器。我们不同意这样做，因为它们不仅不方便，同时很可能带来错误。但我们不阻止你。当你这样做，在保存时，请把文件保存为仅文本类型的(text only)，字处理软件有它自己的格式，这些东西通常是无用的。许多字处理软件很可能提醒你 Perl 程序拼写错误，应当使用更少的分号等等。

某些情况下，你可能在一台机器上书写程序，然后在另一台机器上运行。如果需要这样做，确定传输文件时选的是“文本”（text）模式或“ASCII”模式而非“二进制”（binary）模式。选择这种方法的原因是，不同机器有不同的文本格式。如果不这样做，可能得到不一致的结果。某些版本的 Perl，当检测到行结束符不对时，会中断执行。

1．4．1 一个简单的例子

依据传统，关于计算机语言的书籍，应当以“Hello,world”这个程序开始。下面是其 Perl 版本：

```
#! /usr/bin/perl
  print “Hello,word!\n”;
```

我们假设你已经把它输入到文本编辑器中。(别担心这个程序的含义以及它如何执行。你将很快知道)。可以将它以你喜欢的任何名字命名。Perl 不需要任何特别的文件名字或后缀名，但最好不要使用后缀名◆。有些系统中需要像*.plx*（PerL eXecutable,可执行的 Perl）这样的后缀；可以查看系统上的 release notes 来获得这些信息。

◆为什么最好不要后缀呢？想象写了个给保龄球记分的程序，你告诉所有的饿朋友它被称为 bowling.plx。某一天，你决定用 C 重新写它。你应该仍以相同的名字命名，表示它仍用 Perl 写成？还是告诉他们，它有了个新名字？（噢，请不要把它叫做 bowling.c）。事实上，他们不关心你用什么语言写它，他们只管用。因此，如果当初把它命名为 bowling，你将少许多麻烦事。

也许需要做些事情，让你的系统知道它是可执行程序。需要做什么呢，视你的系统而定。也许你只需把它放在某个特定的路径就行了。(通常你的当前目录就行了)。在 Unix 系统中，你需要用 chmod 命令将程序变成可执行的，可能像下面：

```
$ chmod a+x my_program
```

行首的美圆符号（$）(和空格）是 shell 提示符，可能你的系统上有些不同。也可在 chmod 后使用 755 来代替 a+x 。两种方法都是告诉系统这个文件是一个程序（可执行的）。

现在你可以如下运行它：

```
$ ./my_program
```

命令开始的点和斜线表示在当前路径查找程序。事实上并非在所有情况下都需要，在完全理解它之前，你应当每次都使用它◆。如果运行顺利，这看起来像奇迹。通常，你会发现程序有错误。编辑，再试一次，当然不需要每次都使用 chmod 命令，因为这个文件的权限已经被修改过了。(当然，如果没有正确的使用 chomd 命令，你可能在 shell 中得到不允许操作“permission denied”这样的信息)。

◆简言之，它防止你运行另一个相同名字的程序（shell 内嵌的)。新手的一个普遍错误是把它命名为 test，而许多系统都有这样的程序(shell 内嵌的)。这就是新手运行的为什么不是他们自己程序的原因。

1．4．2 这个程序有些什么？

同任意自由格式语言一样，Perl 通常允许使用任意数量的空白(如空格，制表符，换行符)来使程序易于阅读。但大多数 Perl 程序使用一种标准格式，非常像刚才展示的程序。我们强烈的鼓励你使用缩进格式的程序，使你的程序更易阅读；一个好的文本编辑器能代你完成许多事情。好的注释能让程序易于理解。在 Perl 中，注释由#开始，直到本行结束（Perl 中没有“块注释”（block comments））◆。在本书的程序中我们没有使用大量的注释，因为正文中已经解释了它们，而你自己的程序，应当使用注释。

◆但是有许多伪造的方法。查看 FAQ(在许多情况下，可以用 perldoc perlfaq 来查看)

因此，另一种（看起来，有些奇怪）写“Hello,world”的方法是：

```
#! /usr/bin/perl
        print #这是注释
    “Hello, world!\n”
        ;    #不要这样写代码
```

第一行是特殊的注释。在 Unix 系统中◆，如果文本的第一行前两个字符是“#!”，接着的就是执行下面文件的程序。在本例中，这个程序是*/usr/bin/perl*。

#! 行和程序的可移植性相关，需要找到每台机器的存放地点。幸运的是，通常都被放在*/usr/bin/perl* 或*/usr/local/bin/perl* 中。如果不是这样，则需要找到你自己机器上 perl 的存放地点，然后使用那个路径。在 Unix 系统中，可能使用如下一行找到 perl：

```
#! /usr/bin/env perl
```

如果 Perl 存放的路径不在你的搜索路径上，应当询问你的系统管理员或者某一个和你使用同一台机器的人。

在非 Unix 系统中，传统上把第一行写做 #!perl。至少，它立刻告诉程序的维护者，这是一个 Perl 程序。如果#!行错了，通常会在 shell 中得到一些错误信息。通常是一些意想不到的信息，如文件不存在“file not found”。这不是说没有找到你的文件；而是说 perl 没有在*/usr/bin/perl* 那里（其恰当的地方）。我们应当时这条消息更清晰，但它不是 Perl 而是 shell 给的。（顺便提醒下，不要把 *usr* 写成 *user*，因为发明 Unix 的伙计懒于书写，因此省略了许多字符)。

另一个问题是，你的系统可能根本不支持#!。如果这样，你的 shell（或者别的），可能要自己执行你程序，得到一些让人吃惊的结果。如果不知道这些错误信息，你可以查看 *perldiag* 的用户手册。

“main”程序包含了所有 Perl 语句（不包括子程序，你在后面会看到）。和 C 或 Java 不一样，Perl 中没有“main”程序。

和其它语言不同，Perl 中不需要声明变量。如果其它语言中你必须申明变量，这可能让你惊奇。但它让我们快速写出 Perl 程序。如果程序只有两行，不希望其中一行仅仅是申明变量。如果你想声明变量，这是好事；第四章有详细说明。

许多语句，由表达式后接分号组成。下面是你已经看了几次的语句：

```
print "Hello,world!\n";
```

你可能猜想，这行将打印出 Hello,world!。结尾是\n，如果使用过 C，C++，Java，你可能知道它是换行。当打印出这条信息后，换行，shell 提示符出现在新行上，而不是在上一条信息之后。单行的输出应当以换行符结束。下一章我们将学习更多的关于换行符，和其它由反斜线（\）转义的符号。

1．4．3 怎样编译 Perl？

只需要运行你的 Perl 程序。Perl 的解释器将编译和运行你的程序。

```
$ perl my_grogram
```

当运行程序时，Perl 的内部编译器首先遍历整个源程序，把它转变为内部的字节码，它是程序的一种内部数据结构。Perl 的字节码引擎将运行这些字节码。如果 200 行有一个语法错误，在执行程序的第二行◆，你将得到出错信息。如果某个循环运行 5000 次，它将一次编译；循环将以最快的速度运行。程序的注释不会增加程序的运行开支。如果某个表达式的计算结果是一个常数，那在程序开始运行时，就会以这个常数来替换，而不需每次循环重新计算。

一个可能的例外情形是，当写了一个 CGI 脚本，它可能每分钟被调用成百上千次。(这个使用率很高。如果一天被调用百次，千次，我们并不担心)。许多此类程序都只有很短的运行时间，因此重新调入他们将是笔可观的开支。如果这对你是个严重的问题，你希望找一种方法能让你的程序保持在内存之中。有一个关于 Apache web server (http://perl.apache.org)的模块:CGI::Fast 兴许能帮助你。

可以保存这些编译过后的字节码以减轻编译的负担吗？或者，更好的是，可以把这些字节码转换为别的语言，如 C，然后编译他们？这两件事，在某种程度上都是可行的，但它们可能使程序难于使用，维护，调试和安装，也可能让你的程序运行更慢。Perl 6 在这方面有重大改进，但现在讨论还为时过早（当我们写做此书时）。

1．5 快速了解 Perl

想看一个有些意思的 Perl 程序吗（如果不想，那随便看看）？如下就是一个：

```
#! /usr/bin/perl
   @lines= `perldoc –u –f    atan2`;
   foreach(@lines){
      s/\w<([^>]+)/\U$1/g;
      print;
}
```

当第一次看见这样的 Perl 代码时，你可能觉得很奇怪。(事实上，每次你看到这样的 Perl 代码时，都觉得它们奇怪)。让我们一行一行的来学习它，看看这个例子完成了什么样的任务。(这些解释很简洁；这里只是大致的讲解。在本书的剩下章节，我们将更加详细的讨论它们。现在并不假定你完全理解它，那是以后的事情。)

第一行是#!这一行，我们已经见过了。你也许要修改它，我们已经讨论过了。

第二行运行了一个外部命令，由` `括起来了。(反引号 ` 通常在美式键盘数字 1 的左边。不要和单引号 ‘ 混淆了。) 我们用

的命令的是 perldoc –u –f atan2；在命令行输入这个命令，看看能得到什么结果。perldoc 这个命令能在大多数机器中使用，它显示相关的文档◆。这个命令告诉你一些关于反正切函数 atan2 的信息；在这里我们把它做为一个外部命令，并处理它的输出信息。

◆如果 perldoc 不能使用。那可能是因为你的系统没有命令行接口，你的 Perl 不能通过反引号或 piped-open（请查看 chapter14）来运行命令（如 perldoc）。如果是这样的，应当跳过需要利用 perldoc 的练习。

反引号内的命令的输出被保存在@lines 中。下一行是一个循环，它依行处理@lines 中的信息。循环内语句是缩进的。虽然 Perl 并不需要这样，当这是好的编程习惯。

循环内的第一行让人惊慌；它是 s/\w<([^>]+)>/\U$1\g；这里不过多的讨论细节，我们只提示下，它能改变有特殊标记（<>）的行,在每一 perldoc 这个命令的输出中，都至少有一行具有这样的形式。

下一行，令人惊奇的是，它输出每一行（可能是修改过的）。输出的结果和 perldoc –u –f atan2 类似，但标记（<>）内的内容有些不同。

总结下，通过这几行程序，我们运行了另一个程序，把它的输出保存在内存中，修改内存中的数据，再把结果输出来。这种把数据从一种形式转换成另一种形式的程序在 Perl 中很常见。

1．6 第六节 练习

通常，每一章都会由一些练习来结束，答案在附录 A 中。但是不需要完成这些练习来结束本章的学习，它们只是正文的补充。

如果不能在机器上练习这些习题，询问你附近的专家。记住应当仔细的推敲这些习题。

1. [7]输入“Hello,world”这个程序，让它运行起来。(你可以任意命名，但像 ex1_1 这样的名字就显得好些，它表示第一章第一个练习。)

2. [5]在命令行输入 perldoc –u –f atan2 这个命令，注意它的输出。如果命令无效，询问你的管理员或者从文档（这个版本的 Perl 文档）中查看调用 perldoc 或其等价的方法。(你需要做这些来完成下一个练习)

3. [6]运行第二个例子（前一节中），观察它的输出。(提示：注意正确输入这些标点符号)。注意到和第二题输出的不同地方了吗？

第二章 标量数据

在英语以及许多其它的语言中，需要区别单数和复数。作为一门由语言学家发明的语言，Perl 也是类似的。同一般情况一样，Perl 也有数据类型—标量◆。标量是 Perl 中最简单的数据类型。大多数的标量是数字（如 255 或 3.25e20）或者字符串（如 hello◆或者盖茨堡地址）。你也许把数字和字符串看作不同的事物，但 Perl 几乎以相同的观点来看待它们。

◆这个概念和数学或者物理学中的标量（一个单独的是事物）没有多少关系；Perl 中也没有向量。

◆如果使用过别的编程语言，你可能把 hello 看作 5 个字符的组合，而不是一个单独的东西。但在 Perl 中，一个字符串是一个标量数据。当然，可以使用这个字符串内部的值，你将在后面章节中了解到怎么做。

标量数据可有操作符（如相加和串联），通常会产生一个新的标量数据。标量数据的值可以存放在标量变量中。标量可以从文件或设备读取，也可以写进去。

2．1 数字

虽然标量在大多数情况下不是数字就是字符串，现在我们最好还是将它们分开来看待。我们首先讨论数字，再讨论字符串。

2．1．1 所有数字内部的格式一致

在下面几段中，你将看到整数(如 255，2001 等)和浮点数（有小数点的实数，如 3.14159，$1.35x10^{25}$），但在内部，Perl 都把它们当作双精度浮点数来处理◆。这就是说在 Perl 内部没有整数值。程序中的整数被当做等价的浮点数来处理◆。你也许注意不到这种转换（或者不关心），但你不应当寻找只属于整数的操作符（不能被浮点数使用的），因为它们不存在◆。

◆双精度浮点类型类似于 C 中由 double 定义的类型。它们的大小可能和具体的机器相关，许多当代的系统都使用 IEEE-754 的格式，它有 15 位精度，其范围至少在 1e-100 到 1e100 之间。

◆有时，Perl 也会使用内部的整数，其对程序员不可见。这样做导致的唯一不同是，程序将运行更快。谁又能抱怨它呢？

◆Perl 中有 integer pragma.但如何使用它超出了本书的范围。正如你将看到的，某些操作可以从浮点数得到整数。但那不是我们此刻讨论的问题。

2．1．2 浮点数

数字符号（literal）是 Perl 程序源代码中代替某个值的方法。数字符号不是计算或 I/O 操作的结果，它是直接写进代码中的数据。

你可能已经很熟悉 Perl 的浮点数。有或没有小数点的数字都是允许的(包括+或-号),也可带一个十进制的指数(符号为 E)。

```
1.25
255.000
255.0
7.25e45   #7.25x10 的 45 次方(一个大整数)
-6.5e24   # -6.5x10 的 24 次方（一个大的负数）
-12e-24   #- -12x10 的-24 次方（很小的负数）
-1.2E-23   #指数符号可以大写(E)
```

2．1．3 整数

整数是简单明了的：

```
0
2001
-40
255
61298040283768
```

最后一个读起来有些困难。Perl 允许用下划线来分隔它，因此可以像下面这样书写：

```
61_298_040_283_768
```

它们是相同的值，但形式上有些不同。你可能认为逗号(,)更恰当，但逗号在 Perl 中有其它用途（下一章中将介绍）。

2．1．4 非十进制整数

同许多其它语言一样，Perl 也允许使用非 10 为底的数字。八进制以 0 开头，十六进制以 0x 开头，二进制 0b 开头◆。在十六进制中 A 到 F（或者 a 到 f）分别表示 10 到 15：

◆“前置 0”指示符只对数字有效，对由字符串转换过来得数字无效，在本章后面你可以看到。可以利用 oct()或 hex()把某个看起来像八进制或十六进制的数据串转换成数字。虽然没有“二进制”（bin）函数来转换二进制的值，如果某个字符串以 0b 开头可由 oct()做到。

```
0377        #八进制数字 377，等同于十进制数字 255
0xff          #十六进制数字 FF，等同于十进制数字 255
0b11111111   #等同于十进制数字 255
```

这些数字表面上看起来并不相同，但这三个数在 Perl 中都代表同一个数。对于 Perl 来讲，0 xFF 或 255.00 是没有区别的，因此选择一种你和你的程序维护者（我们是指那个要读懂你代码的可怜伙计。通常，这个可怜的家伙就是你，你很可能想不起 3 个月前，你为什么要那样做）认为最有意义的一种。

当一个非十进制的数字超过 4 位时，读起来将很困难。由于这个理由，Perl 允许你使用下划线来区分：

```
0x1377_0B77
0x50_65_72_7C
```

2．1．5 数字操作符

Perl 除了提供通常的操作符 加（+），减（-），乘（*），除（/）等等之外：

```
2+3          #2+3，5
5.1-2.4      #5.1-2.4，2.7
3*12         #3*12，36
14/2         #14/2，7
10.2/0.3     #10.2/0.3，34
10/3         #通常是浮点除，3.33333……
```

还提供了模数运算符（%）。10%3 的值是 10 除以 3 的余数。两个操作数首先变成它们对应的整数值，如 10.5%3.2 转换为 10%3◆后再计算。另外，Perl 中提供了和 FORTRAN 类似的指数操作符，C 和 Pascal 很希望有类似的能力。这个操作符由两个*号表示，如 2**3，表示 2 的 3 次方，等于 8◆。我们将在需要的地方介绍其它的数字操作浮。

◆注意，在模数运算中，如果有一个操作数为负数，那其结果和 Perl 的具体实现相关。

◆通常不能进行一个负数的非整数次方的运算。对数学有一定了解的读者知道，这将产生一个复数（数学概念中的复数：如 1+2i，译注）。如果想进行类似的预算，你需要 Math:：Complex 这个模块

2．2 字符串

字符串是一串字符（如 hello）。字符串可能是字符的任意组合◆。最短的字符串不含任何字符。最长的字符串，可以填满整个内存。这符合 Perl 的哲学，只要有可能就不加任何内嵌的限制。通常字符串是可打印字符，数字，标点符号的序列(从 ASCII 32 到 ASCII 126)。但，Perl 中字符串可以包含任意字符，意味着利用字符串(string)你可以创建，遍历，操作二进制数据,而利用别的方法可能遇到极大的困难。例如，你可以把要更新的图片或编译好的程序放入一个 Perl 的字符串变量中，做完相应的修改后，再写回去。

◆和 C，C++不同，Perl 中 NUL 字符没有特殊的含义。Perl 能计算长度，不用靠 null 来判断字符串是否结束。

和数字一样，字符串也可由文字符号(literal)来表示，它用来在 Perl 程序中代表某个字符串。有两种类型的字符串：单引号字符串和双引号字符串。

2．2．1 单引号字符串

单引号字符串是由单引号括起来的字符序列。单引号不是字符串的一部分，但 Perl 可以利用它来辨别字符串的开始和结束。除了单引号，或者反斜线（包括换行字符，如果字符串在下一行继续）之外的任何字符都表示它自身。要得到一个反斜线，可以把两个反斜线放在一起；要得到单引号，需要在单引号前加上反斜线：

```
'fred'    #四个字符：f,r,e,d
'barney' #六个字符
''    #空字符（没有字符）
'Don't let an apostrophe end this string prematruely!'
'the last character of this string is a backslash: \\'
'hello\n' #hello 紧跟着反斜线和 n
'hello
there'   #hello,换行，there (共 11 个字符)
'\'\\'   #单引号(')跟着反斜线(\)
```

单引号字符串中的\n 不会被当作换行符来处理，其仅仅是两个字符\和 n。只有在反斜线(\)后面接的是\或单引号'，其才会被当作特殊符号来处理。

2．2．2 双引号字符串

双引号字符串和在其它语言类似。它也是字符的序列，不同点在于，其由双引号括起来的。现在，反斜线可以用来表示控制字符，或者八进制，十六进制数的表示。下面是一些双引号字符串的例子：

```
"barney" #等同于'barney'
"hello world\n" #hello world,换行
"the last character of this string is a quote mark:\""
"coke\tsprite" # coke, a tab(一个制表符), sprite
```

双引号中字符串"barney"和单引号字符串'barney'相同。和数字一样，0377 只是 255.0 的另一种写法。Perl 允许你以一种更有意义的方式来书写。当然，如果想\和之后的字符成为转义字符（如\n 表示新行），应当使用双引号。

反斜线后接不同的字符其含义不同（通常称为：转义字符）。表 2-1 基本上列出了所有的◆双引号中的转义字符。

◆最近 Perl 中引进了 Unicode 转移符，我们这里不演示它们

表 2-1 双引号字符串中的转义符

符号	含义
\n	换行
\r	回车
\t	制表符

\f	formfeed
\b	退格
\a	响铃
\e	escape（ASCII 中的 escape 字符）
\007	任何八进制值（这里是，007=bell(响铃)）
\x7f	任何十六进制值（这里是，007=bell）
\cC	一个控制符（这里是，ctrl +c）
\\	反斜线
\”	双引号
\l	下个字符小写
\L	接着的字符均小写直到\E
\u	下个字符大写
\U	接着的字符均大写直到\E
\Q	在 non-word 字符前加上\，直到\E
\E	结束\L,\E 和\Q

双引号字符串另一个性质是可进行变量内插，这是说当使用字符串时，如果字符串中含有变量名，将由变量的当前值替换它。我们还没有介绍变量，在本章的后面将继续讨论这个问题。

2．3．3 字符串操作符

字符串可由 **.** 操作符连接(是的，只是一个点) 。它不会改变任何字串，就像 2+3 不会改变 2 或 3 一样。串联之后的字符串可供以后使用：

```
“hello” . “world”      # 同于 “helloworld”
“hello” . ‘ ’ . “world” #同于 “hello world”
‘hello world’ . “\n”   #同于 “hello world\n”
```

串联必须由 . 操作符进行。同别的语言不一样，串联可通过把两个放在一起来达到。

一个特殊的操作符是字符串重复操作符(string repetition operator)，由小写的字母 x 表示。这种操作能把操作符左边字符串重复操作符右边数字那么多次：

```
“fred” x 3            # “fredfredfred”
“barney” x  (4+1)     # “barney” x 5, “barneybarneybarneybameybarney”
5 x 4                 #实际上是 “5” x 4, “5555”
```

值得具体讲解下最后一个例子。字符串重复操作符需要一个字符串作为左操作数，因此数字 5 被转变为字符串 “5”（在一节将详细讨论），一个单字符字符串。这个新的字符串被复制 4 次，产生了一个 4 字符的字符串 5555。如将两个操作数的顺序对调下：4 x 5，将得到字符串 44444。这表示字符串重复操作符不是可交换的。

复制次数（右操作数）在使用之前会把它转换为小于等于它的整数（如，4.8 变为 4）。重复次数小于 1 将产生空串（长度为 0）。

2．2．4 数字和字符串之间的自动转换

大多数情况下，Perl 将在需要的时候自动在数字和字符串之间转换。它怎样知道什么时候需要字符串，什么时候需要数字呢？这完全依赖于标量值之间的的操作符。如果操作符（如+）需要数字，Perl 将把操作数当作数字看待。如果操作符需要字符串（如 . ），Perl 将把操作数当作字符串看待。不必担心数字和字符串的区别；使用恰当的操作符，Perl 将为你做剩下的事。

当在需要数字的地方使用了字符串（如，乘法），Perl 将自动把字符串转换为其等价的数字，就像输入的是十进制浮点数一样◆。因此 “12” * “3” 将给出 36。后面的非数字部分和前面的空格将被去掉，如 “12fred34” * “3”将给出 36 而不会用任何提示◆。在极端情形，当一个不含任何数字的字符串将别转换为 0。如，将 “fred”当作数字来使用时。

◆用首字符 0 表示非十进制值对数字有效，对自动转换没有作用。使用 hex() 和 ort() 来转换此类字符串

◆除非你使用了 warnings,我们将很快讨论到。

同样，如在需要字符串的地方使用了数字（如，字符串连接），数字将转换为字符串。例如，如果你想在 Z 后串接 5 乘以 7 的结果◆，可以这样写：

```
"Z" . 5 * 7   #同于 "Z" . 35,或 "Z35"
```

总之，一句话，不用担心使用的是数字还是字符串（大多数情况下）。Perl 将自动转换它们◆。

◆不用担心效率问题。Perl 能记住转换的结果，因此这一步只做一次。

2．3 Perl 内嵌的警告(warnings)

当程序中包含可能的错误时，可以要求 Perl 警告你。运行程序时，可以在命令行中使用 –w 这个参数把警告打开：

```
$ perl –w my_program
```

或者，如果一直都需要警告(warning),可以在 #! 这一行加上-w，如：

```
#! /usr/bin/perl –w
```

这条命令甚至在 non-Unix 系统中也有效，由于在这些系统中通常与 Perl 的具体路径关系不大，因此可如下书写：

```
#! perl –w
```

在 Perl5.6 或之后的版本中，可以使用 pragma 来打开警告(warning).(注意，它对早期的 Perl 版本无效）◆。

◆warnings progma 允许文字上的警告。你可以在 pelllexwarn 的用户手册中找到详细信息。

```
#! /usr/bin/perl
use warnings;
```

现在，如果将'12fred34'当作数字来用，Perl 将警告你：

```
Argument "12fred34" isn't numeric
```

当然，警告通常只对程序员有意义，对普通用户则没什么用处。如果程序员没有看到警告（没使用警告），这并没什么好处。警告除了在某些时候抱怨可能出错外，不会改变程序的行为。如果看到不能理解的警告信息，可以使用 diagnostics pragma, 通过它可以看到更详细的信息。perldiag 的 mangage 中有对短的 warning(警告)和长的 diagnostic（诊断）的描述。

```
#! /usr/bin/perl
use diagnostics;
```

当把 use diagnostics 加入程序后，在每次调入程序时，它好象暂停了一会儿。那是因为 Perl 做了大量的工作（占去大块内存），使在当 Perl 发现错误时，你能迅速的读其文档，如果有的话。这导致了一种对 Perl 程序优化的方法，当不需要读警告信息相关的文档时，将 use diagnostics 去掉。（当然如果能修改程序，把引起警告的原因去掉，那是最好不过了。但只是取消阅读这些文档已经足够。）

另一种优化方法是，在命令行中使用 –M 这个参数，仅当需要 diagnostics 时才用，而不用每次通过修改源代码来决定是否激活 diagnostics：

```
$ perl –Mdiagnostics ./my_program
Argument "12fred34" isn't numeric in addition(+) at ./m_program line 17 (#1)
    (W numeric) The indicated string was fed as an argument to
        an operator that expected a numeric value instead. If you're
    fortunate the message will identify which operator was so unfortunate.
```

我们将指出代码中可能警告的地方。但在现今版本中的警告信息和将来版本可能不同。

2．4 标量变量

变量是保存一个或多个值的容器◆。变量的名字在整个程序中保持不变，但其包含的值可以变化。

◆如你所见，标量变量仅能还有一个值。但其它变量，如数组或哈希（hash）,可以含有多个值。

标量变量可以存放一个标量值。标量变量的名字由一个美圆符号（$）后接 Perl 标识符：由字母或下划线开头，后接字母，

数字，或者下划线。或者说由字母，数字和下划线组成，但不能由数字开头。大小写是严格区分的：变量$Fred 和变量$fred 是不同的。任意字母，数字，下划线都有意义，如：

```
$a_very_long_variable_that_ends_in_1
```

和变量：

```
$a_very_long_variable_that_ends_in_2
```

是不同的。

标量变量在 Perl 中由$开头◆。在 shell 中，当取值时，需要$；赋新值时，不需要$。在 awk 和 C 中，完全不需要$。如果你在这这几种语言中来回切换的话，你很可能经常出错。这是很正常的。（大多数 Perl 程序员推荐在写 Perl 程序时停止书写 shell, awk，C 程序，当然是否采纳，由你自己决定）。

◆按照 Perl 的行话来讲，被称作"sigil"。

2．4．1 选择好的变量名

通常，应当选择能很好描述你的意图的变量名。例如，变量$r 就不如$line_length 描述性强。如果一个变量只在相临几行中使用，那可以取个像$n 这样的名字，但如果变量要在整个程序中使用的话，最好还是仔细的选择变量名。

同样的，仔细的使用下划线可以使变量名更易阅读和理解，特别是维护你程序的人和你有不同的母语背景时。例如，$super_bowl 这个变量就比$superbowl 好些，因为后者可能被理解为$superb_owl。又如$stopid 是$sto_pid（保存一个进程 ID），还是$s_to_pid(将某个东西转变成进程 ID)，或是$stop_id("stop"对象的 ID)，或者仅仅是$stupid 的错误拼写？

大多数 Perl 程序中的变量都是小写的，和你在本书中见到的一样。在少数情况下，使用大写字母。所有字母均大写（如$ARGV）通常表明这个变量有特殊的地方。当一个变量有超过一个单词时，一些人使用$underscores_are_cool 这种形式，另一些人使用$giveMeInitialCaps 这种形式。无论采取那种，请保持风格一致。

当然，变量名字的好坏对 Perl 没有任何影响。如你可以把三个重要的变量命名为：$OOOOOOOOO, $OOOOOOOO, $OOOOOOOOOO，Perl 不会弄错。但是，如果这样，请不要让我们维护你的代码☺。

2．4．2 标量赋值

标量变量最通常的操作是赋值：将值赋给变量。Perl 中的赋值符是等号（和许多语言类似），将等号右边的值赋给等号左边的变量：

```
$fred = 17;            #将 17 赋给变量$fred
$barney ='hello';   #将五个字母的字符串 'hello'赋给$barney
```

```
$barney = $fred + 3;# 将$fred 的值加上三赋给$barney (20)
$barney= $barney*2;#将变量$barney 乘 2 再赋给$barney (40)
```

注意最后一行中$barney 变量使用了 2 次：一次从中取值（等号右边）一次作为赋值的对象（等号左边）。这是合法的，安全的，且普遍使用。事实上它如此常用，以致 Perl 提供了一种简便写法，你将在下一节中了解到。

2．4．3 二元赋值操作符

像$fred=$fred+3（同一个变量在赋值符两边出现）这样的表达式在 Perl 中经常出现（同 C 和 Java 类似），因此 Perl 提供了一种简便的替代方法：二元赋值操作符。几乎每一个二元操作符都有一个等价的二元赋值形式：由这个符号后接等号组成。例如，下面两行是等价的：

```
$fred = $fred + 5; #没有用二元赋值操作符
$fred+=5;          #利用二元赋值操作符
```

下面的也是等价的：

```
$barney = $barney*3;
$barney*=3;
```

上述两例中，变量借助自身而非别的变量来改变自身的值。

另一个常用的赋值操作符是字符串连接符号（.)；其赋值形式为(.=)：

```
$str = str . " ";   #$str 后接空格
$str .= " ";       #同上
```

2．5 print 输出

通常，应该让你的程序有输出，是一个好主意；否则，别人可能认为程序什么事也没做。print()能完成这种工作。它把一个标量参数作为参数，再把它不做修改的输出来。除非做了某些修改，否则其默认的输出是终端（显示器)：

```
print "hello world\n"; #输出 hello world，后接换行符
print "The answer is ";
print 6 * 7;
print ".\n";
```

也可以将一串值赋给 print，利用逗号分开：

```
print "The answer is ",6*7, ".\n";
```

这是列表，但我们还没讨论到列表，这将以后解释。

2．5．1 字符串中标量变量的内插

当一个字符串由双引号括起来时，如果变量前没有反斜线，则变量会被其值内插◆。也就是说字符串中的标量变量◆将被其值替换。

◆这和数学或统计学中的内插含义是不同的

◆还有一些其它的变量类型，在后面章节中将看到

```
$mean = “brontosaurus steak”;
$barney = “fred ate a $meal”;     #$barney 现在是 “fred ate a brontosaurus steak”
$barney = ‘fred ate a’ . $meal;     #同上
```

从上面得知，不使用双引号也可以得到相同的结果。但使用双引号更方便些。

如果一个变量未被赋值，则将使用空值替换：

◆这是一种特殊的未定义值，undef。在本章后面将介绍到。如果开启了警告，Perl 将提示你内插的变量未定义（未初始化）。

```
$barney = “fred ate a $meat”; #$barney 现在是 “fred ate a ”;
```

如果使用的是单独一个变量，是否使用引号没有影响。如：

```
print “$fred”;   #引号是没有必要的
print $fred;      #更好的写法
```

将单独的一个变量使用引号括起来没有错误，但别的程序员可能会笑你◆。变量内插通常也叫做双引号内插，因为它在双引号中（而非单引号）才有效。在某些别的字符串中也可能被内插，遇到它们时再讲解。

◆是的，可能将其值作为字符串而非数字看待。在极少数情况下，是需要引号的。但几乎大多数情况都是浪费笔墨。

在字符串中变量前（$符号前）加上反斜线(\)，变量将不会被内插（替换）：

```
$fred = ‘hello’;
print “The name is \$fred .\n”;     #打印出美圆符号，变量不会被其值替换
print ‘The name is $fred’ . “\n”;    #同上
```

变量名将是字符串中有意义的最长的那一个。这可能在当你需要在某次匹配就替换的情况下出问题。Perl 提供了一种类似于 shell 的分隔符：花括号({})。 用花括号将变量名括起来。或者将字符串分隔成几个部分，再用连接符(.)串起来：

```
$what = “brontosaurus steak”;
```

```
$n = 3;
print “fred ate $n $whats.\n”;              #不是 steaks，而是$whats 的值
print “fred ate $n ${what}s.\n”;            #现在是使用变量$what
print “fred ate $n $what” . “s.\n”;          #另一种方法
print ‘fred ate ’. $n . ‘ ’. $what . “s.\n”;   #一种复杂的方法
```

2．5．2 操作符优先级和结合性

操作符的优先级规定哪部分先进行预算。例如，表达式 2+3*4，是先进行加呢还是先进行乘？如果先进行加，得到 5*4，20。如果先进行乘（同数学课中学到的一样），将得到 2+12， 等于 14。幸运的是，Perl 中的定义和数学上的一样，先进行乘。由此，我们可以说乘法比加法的优先级更高。

可以使用括号来改变优先级。括号中的表达式将首先被计算（和数学课中学到的一样）。因此，如果想加法先进行，可以使用 (2+3) * 4， 得到 20。如果想先进行乘， 可以使用 2 + (3*4)， 当然此时括号不是必需的。

乘法和加法的优先级是比较容易确定的，但字符串连接符和幂运算的优先级就不是那么明显的。恰当的方法是查看 Perl 的优先级表，如表 2-2◆.（一些操作符没在此表中列出，要查看详细信息，可参考 Perl 的用户手册）。

◆C 和 Perl 中都有的操作符有相同的优先级，这对 C 程序员是一个好消息。

表 2-2 操作符的优先级和结合性（由高到低）

结合性	操作符
左	括号和列表操作符的参数
左	->
	++ --(自增和自减)
右	**
右	\ ! ~ + - (一元操作符)
左	=~ !~
左	* / % x
左	+ - . (二元操作符)
左	<< >>
	Named unary operators (-X filetests, rand)
	< <= > >= lt le gt ge(“不等的”)
	= = != <=> eq ne cmp(“相等的”)
左	&
左	| ^
左	&&
左	||
	.. …
右	?:(三元操作符)

右	= += -= .=
左	, =>
	List operators(rightward)
右	not
左	And
左	or xor

上表中，上面的操作符比下面的操作符优先级更高。同一优先级的操作符由结合性来决定计算顺序。

同优先级类似，结合性是用来规定有相同优先级的操作符的计算顺序：

```
4** 4** 2     #4**（3**2）
72/12/3       #(72/12)/3
36/6*3        #(36/6)*3
```

在第一条中，**是右结合的，所以右边的先进行计算。同样的，由于* /是左结合的，所以左边的先进行运算。

那么，需要记住优先级表吗？不需要！事实上没有人那样做。如果记不住优先级时，可以使用括号。毕竟，如果你不知道其优先级顺序，那很可能程序的维护者也记不住。因此，应当善待他/她，因为那个人很可能就是你！

2．5．3 比较运算符

对于数字的比较，Perl 提供了 < <= = = >= != 这些操作符。每一种返回的值为 true 或者 false。在下一节中将了解到更多。其中一些可能和你在别的语言中学到的不一样。例如：= = 相等；= 赋值；!= 不等，因为<>在 Perl 中有别的用途。使用 >= 而非 => 作为“大于等于”，也是由于 => 有其它用途。事实上，绝大多数符号的组合在 Perl 中都是有特殊用途的。

对于字符串比较，Perl 有如下的一些有趣的字符串比较符：lt le eq ge gt ne。它们将一个字符接着一个字符的比较两个串来判断它们的关系：相等，小于，等等。(注意，在 ASCII 中，大写字母在小写字母的前面)。

比较运算符（数字的和字符串的），列在表 2-3 中。

表 2-3 数字和字符串的比较运算符

比较关系	数字	字符串
相等	= =	eq
不等	!=	ne
小于	<	Lt
大于	>	gt
小于或等于	<=	le
大于或等于	>=	ge

下面是一些关于比较运算符的例子：

```
35 != 30+5           #false
35 == 35.0           #true
'35' eq '35.0'       #false  (按照字符串比较)
'fred' lt 'barney'   #false
'fred' lt 'free'     #true
'fred' eq 'fred'     #true
'fred' eq 'Fred'     #false
' ' gt ''            #true
```

2．6 if 控制结构

一旦能比较两个值时，就希望能根据这些比较结果作判断。和别的语言类似，Perl 中也提供了 if 控制结构：

```
if($name gt 'fred'){
    print " '$name' comes after 'fred' in sorted order.\n";
    }
```

如果需要另一种选择，可以使用关键字 else：

```
if($name gt 'fred'){
   print " '$name' comes after 'fred' in sorted order.\n";
}else{
print " '$name' does not come after 'fred'.\n";
print "Maybe it's the same string, in fact.\n";
}
```

花括号是必须的（这一点和 C 不同）。将块中的代码缩进是一个好主意；这样将使代码易于阅读。如果使用的是程序员编辑器（参见第一章），它将为你自动完成许多事。

2．6．Boolean 值

在 if 控制结构的条件判断部分可以使用任意的标量值。这在某些时候将很方便，如：

```
$is_bigger = $name gt 'fred';
if($is_bigger){…}
```

那么，Perl 是怎么判断其值得 true 或 false 呢？Perl 不同于其它的一些语言，它没有 Boolean 类型。它利用如下几条规则◆：

◆事实上 Perl 不是用的这些规则，但你可以利用它们方便记忆，其结果是一致的。

● 如果值为数字，0 是 false；其余为真

● 如果值为字符串，则空串(‘’)为 false；其余为真

● 如果值的类型既不是数字又不是字符串，则将其转换为数字或字符串后再利用上述规则◆。

◆这意味着 undef（很快会看到）为 false。所有的引用（在 Alpaca 书中有详细讨论）都是 true。

这些规则中有一个特殊的地方。由于字符串‘0’和数字 0 有相同的标量值，Perl 将它们相同看待。也就是说字符串‘0’是唯一一个非空但值为 0 的串。

如果想得到相反的值，可以使用一元非运算符 ！。如果其后面的是 true，则得到 false；反之，则得到 true：

```
if(! $if_bigger){
#当$if_bigger 非真时，运行此代码
}
```

2．7 用户输入

现在，可能想你的 Perl 程序怎样才能从键盘上得到输入呢？有一种简单方法：使用行输入操作符(line-input operator)，<STDIN>◆。

◆是行输入运算符对文件句柄 STDIN 的操作。但直到第五章才介绍文件句柄。

<STDIN>作为标量值来使用的，Perl 每次从标准输入中读入文本的下一行，将其传给<STDIN>。标准输入可以有很多种；默认的是键盘。如果还没有值输入<STDIN>，Perl 会停下来等你输入一些字符，由换行符结束（return）◆。

◆坦白讲，是你的系统等待输入，Perl 等待你的系统。具体的细节与机器和配置有关。由于是系统而非 Perl 控制你的输入，因此要更正错误的输入通常可以在按下回车前使用退格键（backspace）。如果想更多的控制输入，可以使用 Term::ReadLine 这个模块，在 CPAN 中可以下载到。

<STDIN>中的字符串通常由一个换行符作为结尾◆。因此，可以如下操作：

◆例外的情况是，标准输入流在行中间就结束了。当然，普通的文本文件通常不是这样。

```
$line = <STDIN>;
if($line eq “\n”){
    print “That was just a blank line!\n”;
}else{
    print “That line of input was: $line”;
}
```

实际上，通常你不需要保留换行符，因此需要 chomp 来去掉它。

2．8 chomp 操作

第一次读到 chomp 函数时，它看起来过于专门化。它对变量起作用，而此变量含有字符串。如果字符串结尾有换行符，chomp 可以去掉它。这基本上就是它能完成的所有功能，如下例：

```
$text = “a line of text\n”;    #也可以由<STDIN>输入
chomp($text);                  #去掉换行符(\n)。
```

它非常有用，基本上你的每一个程序都会用到它。如你将知道，这是将字符串末尾换行符去掉的最好方法。基于 Perl 中的一条基本原则：在需要使用变量的地方，可以使用赋值表达式来代替。我们有更简单的使用 chomp 的方法。Perl 首先做赋值运算，再使用这个变量。因此使用 chomp 的最常用方法是：

```
chomp ($text = <STDIN>);   #读入，但不含换行符

$text = <STDIN>;
chomp ($text);             #同上，但用两步完成
```

第一眼见到时，第一种组合的方法看起来复杂些。如果把上述其看成两步操作，读一行再 chomp，那写成两个语句的方法看起来自然些。如果将其看作一个操作，读入一行但不包括换行符，那写成一个语句的方法更恰当。由于绝大多数 Perl 程序员使用第一种写法，你也应该使用它。

chomp 是一个函数。作为一个函数，它有一个返回值，为移除的字符的个数。这个数字基本上没什么用：

```
$food = <STDIN>;
$betty = chomp $food;   #得到值 1
```

如上，在使用 chomp 时，可以使用或不使用括号（）。这又是 Perl 中的一条通用规则：除非移除它们时含义会变，否则括号是可以省略的。

如果结尾有两个或两个以上的换行符◆，chomp 仅去掉一个。如果没有，那什么也不做，返回 0。

◆这种情况在一次读入一行时不会发生，但使用了输入分隔符(input separator)（$/）（其不为换行符(\n)），read 函数，或者将一些字符串结合起来就有可能发生。

2．9 while 控制结构

和许多的程序语言一样，Perl 也提供了循环结构◆。while 语句可以循环执行其内部的一块代码直到其条件非真：

◆基本上每个程序员都有创建过无限循环语句的经历。如果程序不停的运行，你可以像关闭系统中别的程序那样来关闭 Perl 程序。通常是使用 CTRL+C；检查你的系统文档来了解具体的信息。

```
$count = 0;
while ($count < 10) {
    $count + = 2;
    print “count is now $count\n”; #打印出 2  4  6  8  10
}
```

条件中真假值的判断和 if 结构中是一样的。和 if 控制结构相同，花括号是必须的。判断条件在迭代前执行，如果条件为假，则一次也不执行。

2．10 undef 值

在变量被赋值之前使用它会有什么情况发生呢？通常不会有什么严重的后果。变量在第一次赋值前有一个特殊值 undef，按照 Perl 来说就是："这里什么也没有，请继续"。如果这里的"什么也没有"是一些"数字"，则表现为 0。如果是"字符串"，则表现为空串。但 undef 既非数字也非字符串，它是另一种标量类型。

由于 undef 在需要数字的地方可以自动转化为 0，因此可以如下的写代码：

```
#将一些基数相加
#n = 1;
while($n < 10){
    $sum += $n;
    $n +=2;#下一个奇数
}
print “The total was $sum.\n”;
```

上述代码在$sum 未初始化(undef)时也能正确执行。第一次执行时，循环体中第一行$n 值为 1，因此将 1 加给$sum。而$sum 就像已经有值 0，因为$sum 值为 undef。现在其值 1。之后，由于其已被初始化，其过程同普通的类似。

同样的，针对字符串的情形：

```
$string .= “more text\n”;
```

如果$string 为 undef；则是空串后接"more text\n"。反之，则是其值后接 "more text\n"。

Perl 程序员在使用新变量时，经常不初始化，从而将变量作为 0 或者空串使用。

许多操作当参数不恰当时返回 undef。如果没做特殊处理，通常会得到 0 或者空串。实践中，这几乎不会有什么问题。实际上，许多程序员利用这种性质。但应当知道如果警告是打开的，那 Perl 在你不恰当的使用未定义值时会提醒你。例如，将一个 undef 的变量赋给另一个变量不会有什么问题，但如果 print 某个未定义的值则将引起警告。

2．1．1 defined 函数

能返回 undef 的操作之一是行输入操作，<STDIN>。通常，它会返回文本中的一行。但如果没有更多的输入，如到了文件的结尾，则返回 undef◆。要分辨其是 undef 还是空串，可以使用 defined 函数，它将在为 undef 时返回 false，其余返回 true。

◆事实上，从键盘输入，不会有“end-of-file”，但其可重定向到文件中再输入。或者用户可能输入某些键，而系统将其作为 end-of-file 看待。

```
$madonna = <STDIN>;
If ($defined ($madonna)){
   print “The input was $madonna”;
}else{
   print “No input available!\n”;
}
```

如果想声明自己的 undef 值，可以使用 undef：

```
$madonna = undef ;   #同$madonna 从未被初始化一样。
```

2．1．2 练习

答案请参考附录 A：

1. [5]写一个程序，计算半径为 12.5 的圆的周长。圆周长等于 2π（π约为3.141592654）乘以半径。答案为78.5。
2. [4]修改上述程序，用户可以在程序运行时输入半径。如果，用户输入 12.5，则应得到和上题一样的结果。
3. [4]修改上述程序，当用户输入小于 0 的数字时，程序输出的周长为 0，而非负数
4. [8]写一个程序，用户能输入 2 个数字（不在同一行）。输出为这两个数的积
5. [8]写一个程序，用户能输入 1 个字符串和一个数字(n)（不在同一行）。输出为，n 行这个字符串，1 次 1 行（提示，使用“x”操作符）。例如，如果用户输入的是“fred”和“3”,则输出为：3 行，每一行均为 fred。如果输入为“fred”和“299792”,则输出为 299792 行，每一行均为 fred。

第三章 列表和数组

如果把标量认为是 Perl 中的单数的话，如我们在第二章开头讨论的，那列表(list)和数组则可认为是 Perl 中的复数。

列表是标量的有序集。数组是包含列表的变量。在 Perl 中这个两个术语是可以互换的。但严格意义上讲，列表是指数据，而数组是其变量名。可以有一些值（列表）但不属于数组；但每一个数组标量都有一个列表，虽然其可以为空。图 3-1 是一个列表，无论其是否存储在一个数组中。

图 3-1 一个有五个元素的列表

索引	值
0	35
1	12.4
2	“hello”
3	1.72e30
4	“bye\n”

列表中每一个元素都是一个独立的标量值。这些值是有顺序的，也就是说，这些值从开头到最后一个元素有一个固定的序列。数组或者列表中的元素是编了号的，其索引从整数 0 开始◆，依次增一，因此数组或者列表第一个元素的索引为 0。

◆数组或者列表在 Perl 中的索引总是从 0 开始，这和某些语言不同。在 Perl 的早期版本中，是可以改变数组列表初始索引值的（不只对一个数组或列表而是一次针对所有的）。Larry 后来发现这是个错误的功能， 其应用也让人失望。但是，如果你特别感兴趣的话，可以参看 perlvar 用户手册中的$[变量。

由于每一个元素是一个独立的标量值，因此一个列表或者数组可以包含数字，字符串，undef 值，或者任意不同类型的标量值的组合。然而，这些元素的类型通常是一致的，例如关于书名的列表（值均为字符串），关于余弦的列表（值均为数字）。

列表和数组可以包含任意数量的元素。最少含有 0 元素，最多可以填满你的可用内存。这里又体现了 Perl 的设计哲学，“没有不必要的限制”。

3．1 访问数组元素

如果你使用过其它语言的数组，那对于 Perl 可以通过索引值来访问元素的做法不会觉得奇怪。

数组中的元素是由连续整数编了号的，其从 0 开始，每增加一个元素，其索引值加一，如：

```
$fred[0] = “yabba”;
$fred[1]= “dabba”;
$fred[2] = “doo”;
```

数组名字（本例中：fred）和标量是属于完全不同的命名空间（namespace）。同一程序也可以同时包含叫做$fred 的标量变量。Perl 将它们当作完全不同的事物来看待，不会混淆◆。(但维护人员可能混淆，所以最好不要将它们以相同的名字来命名)。

◆语法总是无二义性的；也许有些技巧，但是确定的。

可以在任何◆能够使用标量变量（如$fred）的地方使用数组元素（如$fred[2]）。例如，可以使用上一章介绍的方法来获得数组元素的值，或者改变它。

◆实际上是绝大多数。最明显的例外是 foreach 循环中的控制变量(在本章后面将介绍到)，必须是标量变量。还有些例外，如 print 和 printf 的“indirect object slot”和 “indirect filehandle slot”。

```
print $fred[0];
$fred[2] = “diddley”;
$fred[1] .= “whatsis”
```

当然，下标可以是任何能返回数值的表达式。如果其值不为整数，则自动将其转换为小于它的最大整数值：

```
$number = 2.71828;
print $fred[$number-1]; #和 print $fred[1]一样
```

如果下标超出了数组的范围，则其值为 undef。这和通常的变量情况是一样的，如果没有值存放在变量中，则其为 undef。

```
$blank = $fred [142_857]      #此数组元素未存放值，得到 undef
$blanc = $mel;                #$mel 未存放值（未初始化），得到 undef
```

3．2 特殊的数组索引

如果将一个元素存储在数组最后元素的后面的位置，数组会自动增长的。Perl 没有长度的限制，只要你有足够的内存。如果 Perl 需要创建元素，则其值为 undef。

```
$rocks[0] = ‘bedrock’;      #一个元素
```

```
$rocks[1] = 'slate';           #又一个
$rocks[2] = 'lava';            #又一个
$rocks[3] = 'crushed rock';#又一个
$rocks[99] = 'schist';         #现在有 95 个 undef 元素
```

有时需要知道数组最后一个元素的索引。刚才使用的 rocks 数组，其最后一个元素的索引为$#rocks◆。这和数组中元素的个数是不同的，因为数组中包含元素 0。(换句话说，最后一个元素的索引值要比其实际包含的元素个数少一，译者注)。

◆这种糟糕的语法来源于 C shell。庆幸的是，在实际的代码中并不常见。

```
$end = $#rocks;                        #99,最后一个元素的索引
$number_of_rocks = $end + 1;     #正确，但有更好的方法
$rocks[$#rocks] = 'hard rock';      #the last rock
```

由于经常将$#name 的值作为索引，像上面例子那样，因此，Larry 提供了一种简便方法：数组的负数索引值从最后一个元素开始。但不要认为这些索引是循环的。如果数组有 3 元素，那有效的负数索引值是-1（最后一个元素），-2（中间的元素），-3（第一个元素）。实际上，几乎没有人使用除了-1 之外的其它的负数索引值。

```
$rocks[-1] = 'hard rock';   #完成上例中的一种更简单的方法
$dead_rock = 'rocks[-100]'; #得到 'bedrock',第 0 个元素
$rocks[-200] = 'crystal';   #严重错误(fatal error!)
```

3．3 列表

数组是由括号括起来并且其元素由逗号分隔开的列表。这些值组成了数组的元素：

```
（1，2，3）   #含有 1，2，3 的列表
（1，2，3,）  #同上，最后一个逗号被忽略
（）          #空列表-0 个元素
（1 .. 100） #包含 100 个整数的列表
```

最后一个例子使用了范围操作符（range operator）..，它创建了从左值到右值之间所有值的列表。

```
（1 .. 5）                 #同（1，2，3，4，5）
（1．7.. 5．7）            #同上— 最小值和最大值被转换成整数
（5 .. 1）                 #空列表— .. 中的左值应小于右值，否则为空
（0，2 ..  6，10，12）#同（0，2，3，4，5，6，10，12）
（$m .. $n）              #由$m 和$n 的值决定
（0 ..  $#rocks）         #上节中有$#rocks 的介绍
```

从上面最后两个例子中可以看到，列表中的元素并非必须是常数，也可以是在执行此语句时再计算值的表达式：

```
（$m，17）                #两个值；$m 的当前值，和 17
```

（$m+$o，$p+$q）　　#两个值

当然，列表可以包含任意的标量值，如下面的包含字符串的例子：

（“fred”, “barney”, “betty”, “wilma”, “dino”）

3．3．1 qw 简写

实践表明，字符串的列表（如上例）在 Perl 中经常使用。有一种简便的方法可以不用输入大量的引号而达到类似的功能，那就是使用 qw。

```
qw(fred barney betty wilma dino )    #同上，但输入更少
```

qw 表示 “quoted words”或者 “quoted by whitespace,”这依赖于你问的是谁。无论那种解释，Perl 将它们当作单引号字符串处理，你不能像双引号那样在 qw 中使用\n 和$fred。whitespace（空格，像 spaces,tabs,newlines 等字符串）将被忽略，剩下的组成了列表的元素。由于空格被忽略，所以下面（不常用的）是另一种书写方法：

```
qw(fred
        barney              betty
wilma   dino)               #同上，当看起来有些奇怪
```

由于 qw 是一种引用，因此不可以在 qw 内添加注释。

前面两个例子是用括号作为分界符，但 Perl 允许使用任何标点符号作为分界符。下面是一些常用的类型：

```
qw !  fred   barney   betty   wilma   dino !
qw#   fred   barney   betty   wilma   dino #   #有些像注释
qw(    fred   barney   betty   wilma   dino )
qw{   fred   barney   betty   wilma   dino }
qw[   fred   barney   betty   wilma   dino ]
qw<   fred   barney   betty   wilma   dino >
```

如后面四个例子中显示的那样，有时两个分界符是可以不同的。如果开分界符有一个对应的闭分界符，那对应的“右”分界符则为其闭分界符。

如果要在字符串中使用闭分界符，很可能选择的分界符并不太恰当。如果不想或者不能改变分界符，那可以使用反斜线(\):

```
qw! Yahoo\! Google excite lycos !      #其中一个元素为：字符串 yahoo!
```

同单引号字符串一样，两个反斜线，可以得到一个反斜线。

正如 Perl 格言中所说：　做一件事不只一种方法(“There’s More Than One Way To Do It”),你可能猜想为什么有人需要这些不同的方法。在后面你将看到许多此类的例子，它们非常有用。如我们处理 Unix 中的文件名：

```
qw{
  /usr/dict/words
  /home/rootbeer/.ispell_english
}
```

如果斜线(/)是唯一的分界符时，那么上述例子将变得极其繁琐。

3．4 列表赋值

和标量值类似，列表值也可以赋给变量：

```
($fred, $barney, $dino) = (“flintstone”, “rubble”, undef);
```

左边列表中的每一个变量都得到了一个新值，和利用 3 个赋值语句得到的结果是一样的。由于列表在赋值之前已经建立，因此在 Perl 中可以使用如下的简单方法交换两个变量的值◆：

◆和 C 语言不同，在 C 语言中没有完成此类操作的简单方法。C 程序员通常需要使用临时变量，可能是使用宏(macro)来定义的。

```
($fred, $barney) = ($barney, $fred)   #交换两个变量
($betty[0],$betty[1]) = ($betty[1],$betty[0]);
```

如果变量个数（等号左边）不同于其值的个数（等号右边），将发生什么事情呢？在列表赋值中，额外的值会被自动忽略。因为 Perl 认为，如果需要把值存起来，那应当指明其存储的地方。同样，如果有多余的变量，额外的变量被赋予 undef◆。

◆对于标量变量这是对的。对于数组变量将得到空的列表，在后面将看到。

```
($fred, $barney)  =  qw <flintstone  rubble  slate  granite>; #两个值被忽略了
($wilma,$dino)  =   qw[flintstone];                           #$dino 为 undef
```

现在可以给列表赋值了，可以使用如下的一行代码来创建按一个字符串数组◆：

◆我们假设 rocks 在本语句之前是空的。如果之前的$rocks[7]非空。那，这个赋值语句将不会改变其值。

```
（$rocks[0],$rocks[1],$rocks[2],$rocks[3]）  = qw/talc mica feldspar quartz/;
```

当想引用这个数组时，Perl 有一种简单的写法。在数组名前加@(后没有中括号)来引用整个数组。你可以把他读作 “all of the（所有的）”，所以@rocks 可以读作 “all of the rocks（所有的石头）”◆。其在赋值运算符左右均有效：

◆Larry 声称选择美元符号($)和@符号的原因是，可以分别读做$calar(scalar)和@arry(array)。你如果不能按这种方式来记忆，也无所谓。

```
@rocks = qw / bedrock slate lava /;
@tiny = ();                             #空表
@giant = 1..1e5;                        #包含 100，000 个元素的表
@stuff = (@giant, undef, @giant);       #包含 200，001 个元素的表
```

```
@dino = "granite";
@quarry = (@rocks, "crushed rock", @tiny, $dino);
```

最后一个赋值语句将五个元素（bedrock, slate, lava, crushed, rock, granite）赋给变量@quarry，因为@tiny 没有元素。(特别的是，它没有 undef 这个值，但可以像@stuff 那样明确的指定它。) 还有一点需要注意的是数组名字被其列表值替换。数组不能成为列表的一个元素的原因是数组只能包含标量值，不能包含其它的数组◆。没有赋值的数组变量的值为( )，空表。和未初始化的标量变量为 undef 类似，未被初始化的数组为空表。

◆但在 Alpaca 书中，将介绍一类特殊的变量：引用。通过它可以造出被称为 "lists of lists"(列表的列表)的数据结构，还有一些别的有用或者有趣的结构。即便是那种情况，也不是将一个列表存放在一个列表之中，事实上存放的是其引用。

当将一个数组拷贝到另一个数组时，仍然是列表赋值。如下例：

```
@copy = @quarry; #将一个数组中的值拷贝的另一个数组中
```

3．4．1 pop 和 push 操作

可以使用新的，更大的索引(index）将新值存放在数组的末尾。但实际上，Perl 程序员不使用索引◆。因此，在下面几段中，我们将介绍几种不使用索引来操作数组的方法。

◆当然，我们是在开玩笑，但这个玩笑基于 Perl 的一些事实。数组中使用索引并没有发挥 Perl 的威力。如果使用 pop, push 和类似的操作符以避免使用索引，那你的程序通常会比大量使用索引的情况要快，而且能避免 "差一位(off-by-one)"类型的错误，这类错误通常叫做 "边界值错误"。有时，一个初级的 Perl 程序员（想比较 Perl 和 C 的速度）将针对 C 优化过的排序程序（有大量的索引操作），用 Perl 来直接实现（从而有大量的索引操作），惊讶于它为什么如此慢。答案是，"用小提琴来订钉子不是一个好办法"。

通常将数组类似于栈来使用，在其右边添加或者删除数据。(这是数组中"最后"一个元素，其索引最大)。这些操作经常出现，因此提供了特殊的函数。

pop 操作将数组的最后一个元素取出并返回：

```
@array = 5..9;
$fred = pop(@array);     #$fred 得到 9，@array 现在为（5，6，7，8）
$barney = pop @array;    #$barney gets 8, @array 现在为（5,6,7）
pop @array;              #@array 现在为（5，6）(7 被丢弃了)
```

最后一个例子中，pop 使用在"in a void context"，也就是说没有存放其返回值的地方。这样使用 pop 是合法的。

如果数组为空，那 pop 什么也不做（因为没有元素可以移出），并返回 undef。

你可能已注意到 pop 后可以使用或者不使用括号。这在 Perl 中是一条通用规则：如果去掉括号含义不变，那括号就是可选的◆。和 pop 相反的操作是 push，它可以将一个元素（或者一列元素）加在数组的末尾：

◆受过相应教育的人将发现，这是同义反复。

```
push(@array,0);                     #@array 现在为(5,6,0)
push @array,8;                      #@array 现在为（5，6，0，8）
push @array,1．． 10;               #@array 现在多了 10 个元素
@others =qw/9  0  2  1  0 /;
push @array,@others;                #@array 现在又多了 5 个元素（共有 19 个）
```

push 的第一个参数或者 pop 的唯一参数必须是数组变量。

3．4．2 shift 和 unshift 操作

push 和 pop 对数组的末尾进行操作（或者说数组右边有最大下标的元素，这依赖于你是怎样思考的）。相应的，unshift 和 shift 对一个数组的开头进行操作（数组的左端有最小下标的元素）。下面是一些例子：

```
@array = qw# dino fred barney #;
$m = shift (@array);                #$m 得到 “dino”, @array 现在为(“fred”, “barney”)
$n = shift @array;                  #$n 得到”fred”, @array 现在为 （“barney”）
shift @array;                       #@array 现在为空
$o = shift @array;                  #$o 得到 undef, @arry 仍为空
unshift(@array,5);                  #@array 现在为（5）
unshift @array,4;                   #@array 现在为(4,5)
@others = 1..3;
unshift @array, @others;            #array 现在为（1,2,3,4,5）
```

和 pop 类似，如果其数组变量为空，则返回 undef。

3．5 将数组插入字符串

和标量类似，数组也可以插入双引号的字符串中。插入的数组元素会自动由空格◆分开：

◆分隔符是变量$”的值，其默认值为空格(space)。

```
@rocks = qw{ flintstone slate rubble };
print “quartz @rocks limestone\n”;          #输出为 5 种 rocks 由空格分开
```

插入的数组元素的第一个元素前面和最后一个元素后面不会插入空格，如果需要可以自己加入：

```
print “Three rocks are:       @rocks.\n”;
print “There’s nothing in the parens   (@empty)   here..\n”;
```

如果忘了数组插入的规则，当把 email 地址插入双引号字符串时可能出现意想不到的结果。由于历史原因◆，这将引起编

译时的严重错误：

◆你可能会问：在 Perl5 之前，Perl 将不会替换没有定义过的数组标量。因此，“fred@bedrock.edu” 将表示 email 地址。但当某人加入了一个变量@bedrock；则这字符串将变成 “fred.edu” 或者更糟。

```
$email = “fred@bedrock.edu”;                    #错误！将会替换@bedrock
$email = “fred\@bedrock.edu”;                   #正确
$email = ‘fred@bedrock.edu’;                    #另一种方法
```

因此，在即将发行的 Perl5（现在快发行 Perl6 了：译者注），没有定义的数组变量的行为和没有定义的标量变量行为一致，也就是说，当警告打开时，没有初始化的数组变量将引起警告。这是 Perl 开发者 10 年来经历的错误的结果。

只有一个元素的数组的被其值替换的行为和你预期的类似：

```
@fred = qw(hello dolly);
$y = 2;
$x = “This is $fred[1]’s place”;                # “This is dolly’s place”
$x = “This is $fred[$y-1]’s place”;             #同上
```

索引表达式被当作普通表达式求值,看起来和不在字符串中是一样的。其变量不会首先被赋值的。换句话说，如果$y 为 “2*4”，那上述表达式的值为 1，而非 7，因为 “2*4”首先当作数字时（$y 在数字表达式中）为 2◆。如果想在一个标量变量后接一个左中括号符，那应当在其间加入分隔符，否则会被作为数组看待：

◆当然，当把警告打开时，Perl 会提醒你 “2*4” 是一个 funny-looking 数字。

```
@fred = qw(eating rocks is wrong);
$fred = “right”;                        #我们将打印 “this is right[3]”
print “this is $fred[3]\n”;             #打印出 “wrong”使用 $fred[3]
print “this is ${fred}[3]\n”;           #打印出 “right”（由花括号分开）
print “this is $fred” . “[3]\n”;        #正确（两个字符串，右 . 分开）
print “this is $fred\[3]\n”;            #正确(利用反斜线转义)
```

3．6 foreach 控制结构

如果能处理整个数组或列表，那将是非常方便的，因此 Perl 提供了这种方法。foreach 从列表的第一个元素一直循环执行到最后一个元素，一次迭代一个：

```
foreach $rock (qw/ bedrock slate lava /){
      print “One rock is $rock.\n”;             #打印出 3 种 rocks
      }
```

控制变量（本例中为$rock）每一次迭代从列表中取出一个新值。第一次为 “bedrock”,第三次为 “lava”。

控制变量不是这些列表元素中的一个拷贝而是这些元素本身。也就是说，如果在循环中修改这个变量，那原始列表中的元素也会被修改，如下面代码段所显示。这条性质是有用的，但是，如果不清楚，可能对其结果感到吃惊。

```
@rocks = qw/ bedrock   slate   lava /;
foreach $rocks(@rocks){
  $rock = “\t$rock”;                        #@rocks 的每一个元素前加入一个 tab
  $rock . = “\n”;                           #每一个元素后加一个换行符
}
print “The rocks are:\n”,@rocks;          #每一个元素都被缩进了，并且一个元素占一行
```

当循环结束时$rock 的值为多少呢？其值同循环开始之前相同。foreach 循环中控制变量的值会被 Perl 自动保存和恢复。当循环进行时，是没有办法改变其值的。循环结束时，变量的值会回到循环开始前，如果没有值则为 undef。这意味着如果有一个变量和控制变量有相同的名字：“$rock”，不用担心会混淆它们。

3．6．1 Perl 最常用的默认变量：$_

如果在 foreach 循环中省略了控制变量，那 Perl 会使用其默认的变量：$_。除了其不寻常的名字外，这和普通变量类似，如下面代码所示：

```
foreach(1..10){                          #使用默认的变量$_
   print “I can count to $_!\n”;
   }
```

虽然它不是 Perl 中唯一的默认变量，但无疑是使用的最普遍的。你将在许多例子中看到 Perl 在没有要求它使用某个变量或值时，会自动使用变量$_，这将节约程序员大量的时间来思考使用哪一个变量。为了消除你的疑虑，下面的 print，就是使用$_的一个例子：

```
$_ = “Yabba dabba doo\n”;
print;                          #打印出默认变量$_。
```

3．6．2 reverse 操作

reverse（逆转）操作将输入的一串列表（可能是数组）按相反的顺序返回。如果不喜欢范围操作符：．．，只能从小到大，那可以使用 reverse 来解决这个问题：

```
@fred = 6 ．． 10;
@barney = reverse (@fred);          #得到 10，9，8，7，6
@wilma = reverse 6 ．． 10;         #同上，没有使用额外的数组
@fred = reverse @fred;             #将逆转过的字符串存回去
```

注意最后一行，其中@fred 使用了 2 次。Perl 通常先计算变量的值（赋值号右边），再进行赋值。

注意 reverse 返回逆转的列表，它不会改变其参数的值。如果返回值没有赋值给某个变量，那这个操作是没有什么意义的：

```
reverse @fred;                  #错误，没有改变@fred 的值
@fred = reverse @fred;          #改变了@fred 的值
```

3．6．3 sort 操作

sort 操作将输入的一串列表（可能是数组）根据内部的字符顺序进行排序。如对于 ASCII 字符串，将根据 ASCII 序进行排序。当然，ASCII 中有一些奇怪的地方，如大写字母在小写字符的前面，数字在字符的前面，而标点符号散布在各处。但按 ASCII 排序只是其默认的行为，在第十三章中，可以看到如何按你想要的顺序进行排序的方法：

```
@rocks    = qw/ bedrock   slate   rubble   granite /;
@sorted   = sort(@rocks);               #得到 bedrock, granite, rubble, slate
@back = reverse sort @rocks;            #为 slate 到 bedrock
@rocks = sort @rocks;                   #将排序的值写回@rocks
@numbers = sort 97 .. 102;              #得到 100，101，102，97，98，99
```

从最后一个例子可以看到，如果将数字按照字符串进行排序可能得到没有意义的结果。当然，默认的排序规则下，任何由 1 开头的字符串先于由 9 开头的字符串。和 reverse 一样，其参数是不会受到影响的。如果想将某个数组排序，那必须将排序之后的结果存回数组中：

```
sort @rocks;                    #错误，不会修改@rocks
@rocks = sort @rocks;           #现在@rocks 值是经过排序的
```

3．7 标量和列表上下文

本节是本章中最重要的章节。事实上，是整本书的最重要一节。可以毫不夸张地说，你的整个 Perl 职业生涯都是建立在对本节的理解之上。如果之前的章节你都未认真阅读，那本章千万不能马虎。

但这并非说本节难于理解。本节仅是一个简单的概念：一个给定的表达式在不同的上下文中其含义是不同的。这本身没什么新奇的地方；事实上这在自然语言中是司空见怪的。例如，在英语中◆，假设某人问你 "read"◆的含义。它的具体含义和你怎样使用它有关。除非你知道它使用的上下文环境，否则不知道其确切的含义。

◆如果英语不是你的母语，那这个类比可能对你不太明显。但上下文相关在任何语言中都存在，因此可以在你自己的母语中找到这样类似的例子。

◆抑或它们问的是 "red"的含义，如果他们是嘴上说的。无论哪种情况，其含义都是不确定的。 如 Donglas Hofstadter 所言，没有一种语言可以把每一种思想都明白无误的表达出来，特别是本句话。

上下文是指表达式存在的地方。当 Perl 解析表达式时，它通常期望一个标量值或者列表值◆。这既称为表达式的上下文环

环境◆。

◆当然，也可能是别的类型。还有些别的 contexts 没有在此处给出。没有人知道 Perl 中使用了多少种 contexts；Perl 社区中的首脑人物在这个问题上也没一致答案。

◆这和人类语言中使用的情况类似。如果犯了语法错误，那很快会被发现，因为你知道在特定的地方使用特定的词。最终，你将用这种方法来阅读 Perl，但首先应该以这种方式来思考。

```
42 + something        #something 必须是标量
sort   something      #something 必须是列表
```

如果 something 是相同的字符串，在一种情况下，它返回一个变量值，在另一种情况下，它可能返回列表◆。Perl 中的表达式将根据其 context 返回适当的值。例如，一个数组的“name◆”，在列表 context 中，它返回列表元素；在标量 context 中，它返回数组元素的个数：

◆当然这个列表可能只包含一个元素。也可能为空，或者包含任意数量的元素。

◆数组@people 的真实名字是 people。@只是一个限定词(qualifier)。

```
@people = qw( fred   barney   betty );
@sorted = sort @people;                    #列表 context：barney , betty, fred
$number = 42 + @people;                    #标量 context：42+3，得到 45
```

甚至普通的赋值（赋给标量或者列表）也产生不同的 contexts：

```
@list = @people;   #3 个 People 的列表
$n = @people ;     #数字 3
```

不要得到这样的结果：在标量 context 中能返回元素的个数，在列表 context 中就一定返回这些元素。许多 list-producing 的表达式◆将返回一些更有趣的结果。

◆对于本节，“list-producing”表达式和“scalar-producing”是没有多少区别的。任何表达式都可以产生标量或者列表，这依赖其 context。因此，当我们说“list-producing expressings”,我们是指那些通常在列表 context 中使用的表达式，当其在标量环境中使用时可能产生意想不到的结果（如 reverse 或者@fred）。

3．7．1 在标量 Context 中使用 List-Producing 表达式

有许多表达式通常都是产生列表的。当其在标量 context 中使用时，会得到什么结果呢？让我们看看这个操作的创始人怎么解释的。通常，这个人是 Larry，其文档展现了整个历史。对于 Perl 的学习，大部分是学习 Larry 是怎么的想的◆。因此，一旦你能像 Larry 那样思考时，你就能明白 Perl 的行为。当学习时，你可能需要查看其文档。

◆更准确的说，Larry 创建 Perl 时，是按照你希望它怎样操作来设计的。

一些表达式根本没有标量 context 的值。例如，sort 在标量 context 中返回什么？你不需要要排序一个列表来得到其个数，因此，除非有人按另一种方式实现了 sort，否则其在标量 context 中返回 undef。

另一个例子是 reverse。在列表 context 中，它返回反转的列表。在标量 context 中，返回反转的字符串(或者将反转的结果串成一个字符串)：

```
@backwards = reverse qw / yabba dabba doo /;
#返回 doo, dabba, yabba
$backwards = reverse qw/ yabba   dabba   doo /;
#返回  oodabbadabbay
```

开始时，你可能对于表达式是在标量或者列表 context 中使用不太清楚。但，你将很快习惯它。

下面是一些例子：

```
$fred = something;                    #  标量 context
@pebbles = something;                 #列表 context
($wilma,$betty) = something;          #列表 context
($dino) = something;                  #列表 context
```

不要被只有一个元素的列表所欺骗；最后一个是列表 context 而非标量 context。括号是必须的，它使第四个区别于第一个。如果将其赋给列表（和元素个数无关），则为列表 context。如果赋给数组，也是列表 context。

让我们看看已经出现过的一些别的表达式及其 context。下面是一些标量 context：

```
$fred = something;
$fred[3] = something;
123 + something;
something + 654
if(something){ … }
$fred[something] = something;
```

下面是一些列表 context：

```
@fred = something;
($fred, $barney) = something;
($fred) = something;
push @fred, something;
foreach $fred(something)
sort something
reverse something
print something
```

3．7．2 在列表 Context 中使用 Scalar-Producing 表达式

其用法是显然的：如果一个表达式不是列表值，则标量值自动转换为一个元素的列表：

```
@fred = 6*7;
@barney = “hello” . ‘ ’ . “world”;
```

下面是另一个例子：

```
@wilma = undef;        #OOPS！得到一个元素的列表(undef)，不同于下面的例子
@betty = ();           #将数组置空的正确方法
```

由于 undef 是一个标量值，将 undef 赋给数组不会清空数组。一个更好的方法是将空列表赋给它◆。

◆在现实中，如果变量被合适的定义在恰当的作用域(scope)中，则不需要明确的将其清空。这种赋值语句在优秀的 Perl 程序中很少出现。在下一章将学习作用域。

3．7．3 强制转换为标量 Context

偶尔，你可能需要标量 context 而 Perl 期望的是列表。这种情况下，可以使用函数 scalar。它不是一个真实的函数因为其仅是告诉 Perl 提供一个标量 context：

```
@rocks = qw(talc   quartz   jade   obsidian);
print   “How many rocks do you have?\n”;
print   “I have ”, @rocks, “rocks!\n”;              #错误，输出 rocks 的名字
print “I have ”, scalar @rocks, “rocks!\n”;         #正确，输出其数字
```

奇怪的是，没有对应的函数能强制转换为列表 context。但请相信我们，事实上你并不需要它。

3．8<STDIN>在列表 Context 中

前面学习过行输入操作：<STDIN>，在不同的上下文环境中返回不同的值。像早先描述的那样<STDIN>在标量 context 中返回输入的下一行。在列表 context 中，它将返回这个输入文件的所有剩余部分。而每一行将作为一个独立的元素，如下所示：

```
@lines = <STDIN>; #将输入读入列表 context 中
```

当输入来源于一个文件时，它将读入文件的剩余部分。但如果输入来源于键盘，那文件结束符(end-of-file)是怎样输入的呢？在 Unix 或者类似的系统中，包括 linux, Mac OS X，通常可以输入 CTRL +D◆来表明输入已经结束。Perl 会忽略这个字符，因此它将在屏幕上显示出来。在 DOS/WINDOWS 系统中，使用 CTRL +Z◆。如果你的实际情况和上述不同，查看你系统的文档或者询问你附近的专家。

◆这仅仅是默认值，可以使用 stty 来改变它。但通常是使用它；我们还没有看见某台 Unix 系统不是使用 CTRL+D 来表明 end-of-file 的。

◆在 Perl 的某些 DOS/WINDOWS 版本中可能存在 bug，在第一行中使入 CTRL+Z 可能存在问题。在这样的系统中，可以使用在输入的结束处加上 "\n"来解决。

如果某人运行一个程序，输入了三行，并且使用了恰当的 end-of-file，那其数组将含有 3 个元素。每一个元素为一个由换行符结束的字符串，对应于由 3 个换行符结束的输入。

如果在输入时，能一次去掉(chomp)所有的换行符，岂不是很好？如果将 chomp 应用于一个包含字符串的数组中，他将把这个数组中每一个元素的换行符去掉，如下例所示：

```
@lines = <STDIN>;       #读入所有的行
chomp = (@lines);       #去掉所有的换行符
```

但更常见的做法是：

```
chomp (@lines = <STDIN>); #读入所有的行，不包括换行符
```

当写你自己的程序时，可以采用任意种方法，大多数 Perl 程序员使用第二种，更加紧凑些。

你可能已经意识到这些输入一旦读过，就不能重新读了◆。 到了文件的结尾，就没有更多的数据可以读入。

◆如果是从某个源(source，如文件，译者注)输入,那你可以返回去重新读一次。但这不是这里讨论的问题。

当输入的文件为一个 400MB 的 log 文件时，将发生怎样的情况？行输入操作符读入所有的行，这将占去大量的内存◆。Perl 不会限制你那样做，但你系统的用户（更别提系统管理员）将很可能阻止你。当输入的内容特别大时，应当避免一次将所有的内容都读入内存。

◆一般，实际占去的内存要比文件要大。也就是说，一个 400MB 的文件读入数组时将至少占去 1G 的内存空间。这是因为 Perl 为了节约时间而浪费了些内存。这是一个好的交易。如果内存不够，可以去买一些；但时间不够，那就很难办了。

3．9 练习

1. [6] 写一个程序，将一些字符串（不同的行）读入一个列表中，逆向输出它。如果是从键盘输入的，那在 Unix 系统中应当使用 CTRL+D 表明 end-of-file，在 Windows 系统中使用 CTRL+Z.

2. [12] 写一个程序，读入一串数字（一个数字一行），将和这些数字对应的人名（下面列出的）输出来。（将下面的人名列表写入代码中）。例如，当输入为 1，2，4 和 2，则输出的为 fred, betty, dino, 和 betty：

   fred betty barney dino Wilma pebbles bamm-bamm

3. [8]写一个程序，将一些字符串（在不同的行中）读入一个列表中。然后按 ASCII 顺序将它们输出来。也就是说，当输

入的字符串为 fred, barney, wilma, betty，则输出为 barney betty fred wilma。分别在一行或不同的行将之输出。

第四章 子程序

你已经见过一些内嵌的函数了，如 chomp, reverse, 和 print。但，如其它某些语言一样，Perl 也可以构造子程序(subroutines)，它们是用户定义的◆。这可以让我们在程序中重复使用某段代码◆。子程序的名字是 Perl 中的另一个标识符(由数字，字符，下划线但不能由数字开头的串组成)，有时由可选的符号(&)开头。关于何时需要，何时不需要&的规则，在本章的结尾处将介绍。现在，如果允许时我们都将使用它(&)，这通常是一种安全的做法。当然，在禁止使用时我们会提醒你。

◆在 Perl 中，我们不像 Pascal 程序那样严格的区分：函数(functions)，将返回值；和子程序(procedures)，不返回值。子程序(subroutine)总是用户定义的，而函数(function)可能不是。也就是说，某个函数可能是子程序（subroutine），或者是 Perl 内嵌的函数（funcitions）。这也就是我们把本章叫做 Subroutines 的原因：因为是关于你自定义的而非系统内嵌的函数。

◆本书中 40%的代码来源于公开的商用程序，如果合理使用，75%的代码都可以在你的程序得到重用。

子程序的名字属于一个独立的名字空间，因此 Perl 不会在你的子程序为&fred 同时也有一个变量为$fred 的情况下混淆，当然通常是没有理由这样命名的。

4．1 定义一个子程序

要定义自己的子程序，使用关键字 sub,子程序的名字（无&这个符号），组成子程序的缩进的代码块(花括号中)，如：

◆恰当的说，花括号也属于块的一部分。Perl 不需要缩进块中的代码，但维护人员需要。因此请遵守这个规则。

```
sub marine {
        $n + = 1;   #全局变量$n
        print "Hello, sailor number $n!\n";
  }
```

子程序（本章中若无特殊说明,子程序均指 subroutine,译者注）的定义可以在程序的任意位置，但具有如 C 或者 Pascal 背景的程序员通常将它们放在程序的开头。某些其它的程序员可能将它们放在结尾，以使程序的主要部分在开头出现。不需要在定义之前有任何声明。子程序的定义是全局的；没有某些强大的技巧，Perl 中没有私有子程序(private subroutines) ◆。如果两个子程序有相同的名字，那后一个将覆盖前一个◆。这被看作是一种不好的编成习惯，它将迷惑你的维护人员。

◆如果想使用特殊的技巧，阅读 Perl 的关于私有变量(private variables)中的 coderefs 文档。

◆将会引起警告

如前面例子那样，可以在子程序中使用全局变量。事实上，到现在为止所出现的所有变量均是全局的；也就是说，它们在程序的任意部分都可以被访问。这让纯粹的语言学家感到恐慌，但一些 Perl 的开发者却大量使用它们。在本章后面的“子

程序的私有变量”一节中将介绍怎样创建私有变量。

4．2 调用子程序

可以使用子程序的名字（带有&）来调用子程序◆：

◆通常有括号，即便参数为空。子程序将继承调用者的@_的值，这会马上讨论。因此不要在这里停止，否则程序可能和你预期的行为不同。

```
&marine;      #输出 Hello, sailor number 1!
&marine;      #输出 Hello, sailor number 2!
&marine;      #输出 Hello, sailor number 3!
&marine;      #输出 Hello, sailor number 4!
```

通常，我们说调用(invocation)是指调用子程序(call the subroutine)。

4．3 返回值

子程序可以被某个表达式调用，无论此表达式的值是否被利用。如早些的例子&marine，我们得到此表达式的值，但其值被扔掉了。

许多时候，当调用某个子程序时，需要使用其返回值。这意味着应当注意子程序返回值。所有的 Perl 子程序都会返回值，在 Perl 中返回值和不返回值是没有区别的。当然，不是所有 Perl 子程序返回的值都是有用的。

由于所有的被调用的子程序都要返回值，因此使用特殊的返回值语法在大多数情况下是一种浪费。因此 Larry 将之简化了。当 Perl 遍历此子程序时，将会计算每一步的值。此子程序中最后计算的值将被返回。

例如，下面的子程序：

```
sub sum_of_fred_and_barney{
    print “Hey, you called the sum_of_fred_and_barney suroutine!\n”;
    $fred + $barney;      #返回值
  }
```

子程序中最后一个被计算的表达式为$fred + $barney，因此$fred 和$barney 的和将被返回。下面是一些调用代码：

```
$fred = 3;
$barney = 4;
$wilma = $sum_of_fred_and_barney;           #$wilma 得到 7
print “\$wilma is $wilma.\n”;
$betty = 3 * &sum_of_fred_and_barney;       #$betty 得到 21
print “\n$betty is $betty.\n”;
```

上述代码将得到下面的输出：

```
Hey, you call the sum_of_fred_and_barney subroutine!
$wilma is 7.
Hey, you call the sum_of_fred_and_barney subroutine!
$betty is 21.
```

上述 print 语句是用于调试的，从它可以看到子程序被调用了。当程序调试好后，应当把它们去掉。假设在上述子程序中加入了下面的一行代码：

```
sub sum_of_fred_and_barney{
    print "Hey, you called the sum_of_fred_and_barney subroutine!\n";
    $fred + $barney;                    #其值不会被返回
    print "Hey, I'm returning a value now!\n";    #oops!
}
```

在上述例子中，最后一个被计算的表达式不是$fred + $barney；而是 print 语句。其返回值通常为 1，意思是 "print was succfessful（打印成功）" ◆；但这不是你想要的返回值。因此当向子程序加入新的代码时要小心些，因为返回值为最后一个被计算的表达式。

◆当 print 成功时返回 true，失败时返回 false。下一章将了解到更多种类的失败。

因此，第二个子程序中$fred + barney 的值去哪里了呢？由于没有把它存放在任何地方，Perl 将它丢弃了。如果打开了警告，Perl（注意到两个变量相加但其值未被使用）可能给出如下的警告信息"a useless use of addition in a void context." 术语 void context 是说其值未被存于变量或者被别的方式使用。

"The last expression evaluated"的含义是指最后一个被求值的表达式，而非程序的最后一行。例如，下面的子程序返回$fred 和$barney 的较大值：

```
sub larger_of_fred_or_barney {
      if ($fred > $barney){
            $fred;
      }else{
            $barney;
      }
}
```

最后一个被求值的表达式是$fred 或者$barney，因此其中的某个变量成为返回值。在代码实际运行前，不知道是$fred 或 $barney 的值将被返回。

上述例子没有什么价值。如果每次调用子程序时可以传递不同的参数比依靠全局变量要好许多。这将在下面进行介绍。

4．4 参数(Arguments)

如果子程序 larger_of_fred_barney 不是必须使用全局变量$fred 和$barney，那将变得更有用。如果要得到$wilma 和$betty 中较大的值，现在必须将将它们的值拷贝到$fred 和$barney 中，然后再调用 large_of_fred_or_barney。如果不希望改变这些变量的值，那首先必须把这些变量拷贝到其它变量之中，如$save_fred 和$save_barney。然后，当子程序结束时，再把它们拷贝回变量$fred 和$barney。

幸运的是，Perl 子程序可以带参数。将参数列表传给子程序中的方法是，在程序名后面接括号，括号内存放参数列表，如：

```
$n = &max(10,15);              #此子程序有 2 个参数
```

此参数列表被传到子程序中；这些参数可以被子程序使用。当然，这些参存放在某个地方，在 Perl 中，会自动将此参数列表（此参数列表的另一个名字）自动存放在一个叫做@_的数组中。子程序可以访问次数组变量来确定此参数的个数以及其值。

这也就是说此子程序参数的第一个值存放在$_[0]中，第二个存放在$_[1]，依次类推。但必须强调的是这些变量和$_这个变量没有任何关系，如$dino[3](数组@dino 的一个元素)和$dino 的关系一样。这些参数必须存放在某个数组变量中，而 Perl 存放在@_这个变量中。

现在，可以写子程序$max，其功能类似于&larger_of_fred_or_barney，但不是使用$fred，而使用（$_[0]）；不使用$barney，而使用($_[1])。因此可以如下这些写代码：

```
sub max{
     #和  &larger_of_fred_or_barney 比较
    If($_[0] > $_[1]){
      $_[0];
      }else{
      $_[1];
    }
}
```

我们说，你可以这样做。但使用这些下标有些难看，并且难于阅读，书写，检查，调试等。后面有更好的方法。

这个子程序有另一个问题。&max 这个名字很简单，但当调用的参数不是两个时，此程序不能提示出了错误：

```
$n = &max(10,15,27);    #oops!
```

额外的参数被忽略了；因为此子程序不会使用$_[2]，Perl 不会关心是否有多余的变量。参数不够时也会被忽略，当传入的参数个数不够时，不够的参数会得到 undef 这个值。在本章后面，将会有更好的方法，它可以在参数个数任意的情况下正常工作。

@_是子程序的一个私有变量◆；如果有一个全局变量@_，它将在此子程序调用前存储起来，当子程序调用完成后，其早期的值会被重新赋还给@_◆。这意味着当将参数传递给子程序时不用担心它会影响此程序中其它子程序的@_这个变量的

值。嵌套的子程序调用时，@_的值和上述类似。甚至此子程序递归调用时，每一次调用将得到新的@_，因此子程序调用时将得到其自身的参数列表。

◆除非调用的子程序前有&而后面没有括号（或者没有参数），此时@_从此调用者的上下文(context)得到。这通常不是个好主意，但有时很有用。

◆你可能意识到这里使用的机制和上一章中 foreach 循环是一样的。在每种情况下，变量值被 Perl 自动保存和重新赋值。

4．5 子程序中的私有变量

Perl 在每一次调用时提供@_这个私有变量，那它可以给我们提供私有变量吗？答案是，能。

默认情况下，Perl 中所有变量都是全局的；也就是说，这些变量可以在程序的任意部分使用。你也可以任意时候使用 my 创建私有变量：

```
sub max {
    my($m,$n);                    #新的，私有变量
    ($m,$n) = @_;                 #赋值
    if($m > $n) {$m} else{$n}
  }
```

这些变量是此代码块中的私有的（或者局部的）变量；别的$m 和$n 不会影响此处的两个变量。而且，其它地方的代码不能访问或者修改这些变量◆。你可以将此子程序放在任意地方而不用担心它们会和程序其它地方的$m，$n（如果有）变量混淆◆。在 if 代码块内部，其语句没有分号。Perl 允许省略括号中最后一条语句的分号，在实际代码中，通常仅当此代码块仅包含一条语句时才省略此分号。

◆高级的程序员知道在程序外部可以通过引用而非变量名字来访问私有变量。

◆当然，如果此程序中还有一个叫做$max 的子程序，那将引起混淆。

前一例中的子程序可以变得更简单。你注意到列表($m,$n)书写了 2 次吗？my 操作符可以用在有括号的变量的列表中，因此，通常将上述两个语句如下书写：

```
my($m,$n) = @_;
```

上面的一条语句创建了一些私有变量并给它们赋值，第一个为$m，第二个为$n。几乎每一个子程序都由类似的语句开头。当看见那一行时，你就知道此子程序需要 2 个变量，在此子程序中非别被叫做$m 和$n。

4．6 参数列表的长度

在实际的 Perl 代码中，传递给子程序的参数个数是没有限制的。这符合 Perl 的设计哲学“没有不必要的限制”。当然，有

一点和传统的程序设计语言很不相同，这些语言中要求其子程序中参数个数和参数类型是严格确定的。Perl 的这种灵活特性很好，但（你将在&max 这个函数中看到）可能引起问题，如使用了错误的参数个数来调用子程序。

当然，子程序可以容易的检测@_是否含有恰当的个数。例如，我们可以如下的书写&max◆：

◆当在下一章学习 warn 后，你可以使用恰当的警告信息。如果认为这是个严重的问题，也可以使用 die，这也将在同一章中有介绍。

```
sub max{
    if(@_!=2){
    print “WARNING! &max should get exactly two arguments!\n”;
    }
 #continue as before….
 .
 .
 .
 }
```

if 语句检测参数个数是否恰当（此时，数组是在标量环境中使用的，它将返回其个数），在第三章中有介绍。

但在实际的 Perl 程序中，很少使用这样的方法；更好的方法是子程序可以在任意参数个数的调用者中正常工作。

4．6．1 更好的&max 程序

现在重写&max，使它可以在任意参数个数下都能正常工作：

```
$maximum = &max(3,5,10,4,6);

sub max {
  my($max_so_far) = shift @_;
  foreach (@_){
      if($_>$max_so_far){
         $max_so_far=$_;
       }
  }
  $max_so_far;
  }
```

这段代码使用了一种叫做 high-water mark 的算法：洪水之后，会在岸边留下痕迹。最高处的标记，表明了到达的最高水位。在此程序中，$max_so_far 记录最高的水位。第一行中置$max_so_far 为 3（第一个参数），它是从参数数组：@_中移出来的。此时，@_变成了（5，10，4，6），因为 3 已被移出。此时出现过的最大数为 3。

现在，foreach 循环遍历@_剩下的值。循环中的控制变量是默认变量$_。（请记住，$_和@_没有任何关系；它们名字相同

完全是巧合)。第一次循环时，$_为 5。if 测试语句中发现它比$max_so_far 大，因此$max_so_far 现在变成了 5。

下一次循环中，$_为 10。它是一个更大的值，此时$max_so_far 为 10。

接着，$_为 4。现在$_小于$max_so_far：10。因此，if 中的语句块被跳过。

接着，$_为 6，基于同样的原因，if 语句块被跳过。这是最后一次循环。

现在，$max_so_far 的值将被返回。他为最大值：10。

4．6．2 空参数列表

经过改进后的&max 有了很大的提高，其参数可以不是 2 个。但，如果参数个数为 0，会发生什么情况呢？

初次遇到这个问题时，可能认为其过于深奥。毕竟，谁会调用&max 而不给其参数呢？但是，某人可能如下书写：

```
$maximum = &max(@numbers);
```

而@numbers 可能为空，例如可能从空文件读入。因此，你需要知道，&max 此时会怎样处理呢？

子程序第一行将@_的第一个元素赋给$max_so_far。这不会有什么坏影响；数组为空，shift 操作返回 undef 给$max_so_far。

现在进入 foreach 循环体，由于为空，循环体一次都不执行。

现在，Perl 返回$max_so_far 的值 undef。在某种意义下，这是正确的答案，因为空表中没有最大的元素。

当然，当调用此子程序时注意返回值可能为 undef，或者保证调用的参数非空。

4．7my 变量的注释

这些局部变量可以在任意块中使用，不仅仅是子程序中。例如，可以在 if，while，foreach 等块中使用：

```
foreach (1..10){
    my($square) = $_*$_; #本循环中的私有变量
    print “$_ squared is $squrare.\n”;
    }
```

变量$square 是私有的，仅在此块中可见；在本例中，此块为 foreach 循环块。如果没有大括符（{}），它将是整个文件中的私有变量。到现在为止，你的程序不会超过一个文件，因此不讨论这个问题。只有和它处于一个代码块中的语句才能使用它。这非常有利于代码的维护，如果$square 值错了，则其被限制在相应的代码段中。有经验的程序员（通常是通过辛苦的实践）发现将变量的作用域限制在一页，或者几行代码中，能加速开发和测试。

当然，my 操作不会改变赋值参数的 context：

```
my ($num) = @_;          #列表 context, 同($sum) = @_;
my $num = @_;            #标量 context，同$num = @_;
```

在第一个例子中，$num 得到第一个参数，因为其在列表 context 中；第二个例子中，将得到参数的个数，因为是在标量 context 中。两条语句都可能是正确的；不能仅凭它来判断是否使用错误，因此 Perl 不能提示（warning）你，如果你使用错误。（当然，不能将这两天语句写在同一个子程序中，因为不能在一个作用域内重复定义某个变量）。当读到这样的代码时，可以通过上下文来判断其 context。

记住，如果没有使用括号，my 仅定义一个变量◆：

◆通常，将 warning 打开会报告这样使用 my，或者你自己调用 1-800-LEXICAL-ABUSE 来报告。使用 strict pragma（将在后面提及），将会阻止这类错误的发生。

```
my $fred, $barney;     #错误！没有定义$barney
my ($fred, $barney); #两个均定义了
```

当然，可以使用 my 创建新的，私有数组◆：

◆或者 hashes，在第六章中有介绍

```
my @phone_number;
```

如果新的变量没被赋值的话：标量变量会自动赋与 undef，而数组变量会赋与空列表。

4．8 使用 strict Pragma

Perl 是一种宽容的语言◆。你可能希望 Perl 严格一些；那可以使用 strict pragma。

◆你可能还没注意到。

pragma 可以提示编译器，告诉它一些关于这块代码的信息。如，使用 strict pragma 告诉 Perl 的编译器，应当严格的检查这段代码。

为什么要使用这种方法呢？假设你写一个程序，输入了如下这一行：

```
$bamm_bamm = 3; #Perl 自动创建这个变量
```

现在，你接着输入一些新的代码。当上面一条语句滚动到屏幕以外时，你输入了下面这条语句：

```
$bammbamm + =1; #Oops!
```

由于 Perl 把它当作一个新的变量（下划线是有含义的），它将创建这个变量，并增 1。如果你足够幸运或者聪明，打开了警告(warnings)，Perl 将告诉你这个，或者这两个变量仅使用了一次。如果你没那么幸运，每个变量使用了不止一次，那 Perl 将不会警告你。

告诉 Perl 进行更严格的语法检测，需要在程序顶端 use strict（或者在任意块或者文件中，如果你需要在此部分使用它）：

```
use strict;   #迫使采用更严格的检测
```

现在，除了其它限制外◆，Perl 将要求你在申明每一个新的变量时，使用 my◆：

◆要想知道更多的限制，可以查看关于 strict 的文档。文档是按照 pragma 的名字来归档的。因此，命令 perldoc strict(或者你系统中查看文档的命令)将为你找到相应的文档。

◆当然，还有一些别的声明变量的方法。

```
my $bamm_bamm=3 ; #新的局部变量
```

如果，你把它拼写成别的形式，Perl 将警告你，你没有定义如$bammbamm 的变量，因此这类错误在编译时既能发现。

```
$bammbamm+=1; #没有这样的变量；编译时错误
```

当然，这只是针对 Perl 的新的变量；对于 Perl 内嵌的变量如，$_，@_则不需要申明◆。如果在以前的程序中使用 use strict，那很可能得到许多警告信息，因此你应当养成使用 strict 的习惯。

◆在某些情况下，不应当申明$a, $b，因为它们被 *sort* 使用。因此，遇到这种问题时，使用其它的变量名。事实上，use strict 不会阻止这两个变量，在 Perl 中不算 bugs。

许多人推荐如果程序长度大于一个屏幕，则需要使用 use strict。我们赞成这种观点。

从现在开始，我们例子中的大多数程序（不是全部）将使用 use strict，虽然有时我们没有将它明显的写出。因此，当定义新变量时，我们将使用 my。虽然，我们不经常这样做，但是我们推荐你在你的程序中尽可能的使用 use strict。

4．9 返回操作

返回操作将立刻的从子程序中返回一个值：

```
my @names = qw /fred barney betty dino Wilma pebbles bam-bamm/;
my $result = &which_element_is(“dino”, @names);

sub which_element_is{
  my($what, @array) = @_;
  foreach(0..$#array){ #@array 元素的索引
      if($waht eq $array[$_]){
```

```
      return $_;                #找到既返回
    }
  }
  -1;                        #没有找到元素（此处是可选的）
}
```

这个子程序是寻找@names 中 “dino”的索引值。首先，定义了 2 个参数：$what，查找的对象；@array，被搜索的数组。本例中为@names 的一份拷贝。foreach 循环遍历数组@array（第一个索引值为 0，最后一个为$#array，第三章中有介绍）。

每一次循环时，将检测数组@array 当前的元素的是否为$what 中的元素◆。如果相同，则返回此时的索引值。从当前代码返回的最常用方法是使用 return 语句，它将立刻返回，而不会执行其余的代码。

◆注意此时的比较符为，eq，而非 ==。

如果不存在所要查找的元素呢？在本例中，子程序返回-1，表明 “值不存在”。如果想更加 Perlish，那可以返回 undef，但本例中，程序员选择了-1。上例中使用 return -1 也是正确的，但 return 是多余的。

某些程序员喜欢每次都使用 return 语句，表明其为返回值。例如，你可能想从子程序中立刻返回（非最后一行）而使用 return，如本章前面的&larger_of_fred_or_barney 这个例子。这并非是必须的，但也没什么坏的影响。对大多数 Perl 程序员来讲，这仅是多输入七个字母而已。

4．9．1 省略符号&

有几条规则确定在调用子程序时是否可以省略掉&。如果编译器在调用之前知道此子程序的定义，或者 Perl 从语法中能知道这是一个子程序调用，则子程序前的符号&是可以省略的，像使用内嵌(built-in)函数一样（这些规则后有一个特例，将在下面了解到）。

这意味着如果 Perl 能通过语法就能判断其是否为子程序调用，那将非常方便。例如，如果用括号将一些参数括起来，那其为函数◆调用：

◆在下例中，其调用的子程序为&shuffle。但它可能是一个内嵌的函数。

```
my @cards = shuffle(@deck_of_cards);   # &是不必要的
```

如果 Perl 内部的编译器知道此子程序的定义，则可以省掉其参数的括号：

```
sub division{
  $_[0] / $_[1];         #第一个参数除以第二个参数
  }
my   $quotient = division 355, 113;       #可以省略掉括号
```

上述代码能正常运行，因为如果加上括号不能改变其含义，那括号就是可选的。

但不要将上述子程序的声明放在调用函数的后面，否则编译器不知道调用 division 是什么意思。编译器需在调用前知道其定义，那样就可以将它当作内嵌(built-in)的函数看待了。

上述并非其特别的地方。特别的地方为：如果子程序和 Perl 一个内嵌程序同名，则必须使用&来调用它。编译器将在调用之前检查其定义，而非直接将它当作内嵌的函数来处理。加上&,可以确保你调用了此子程序；不加，则仅当没有同名的内嵌函数时才能调用到它：

```
sub chomp {
    print “Munch, Munch!\n”;
    }
&chomp;   #此处的&是必须的
```

如果没有&，我们将调用内嵌的函数 chomp，虽然已经定义了子程序&chomp。因此，实际的使用规则是：除非知道 Perl 所有的内嵌函数，否则函数调用时都应当使用&。这即是说在你头几百个程序都应当使用&。但如果你看到某人的代码中省略掉了&，这并非一定是错误；可能他们知道 Perl 没有具有相同名字的内嵌函数◆。当程序员打算调用其自己的子程序和调用内嵌的（built-in）函数一样时，通常是写模块（modules)，经常使用原形（prototypes）来告诉 Perl 预期的参数类型。创建模块是一种高级技术；当你准备好时，可以参看 Perl 的关于子程序原型(subtoutine prototypes)和创建模块(making modules)的文档（特别是 perlmod 和 perlsub 两份文档）。

◆当然，他们也可能犯错误；你可以搜索 perlfunc 和 perlop 的用户手册来查看其是否为某个内嵌函数的名字。当把警告（warnings）打开时，Perl 会提示你这种问题。

4．10 非标量返回值

标量并非子程序返回的唯一类型。如果你在列表 context 中调用某个子程序◆，则其会返回列表值。

◆你可以使用 wantarray 函数来判断一个子程序是在标量还是列表 context 中使用的，这可以让你轻松的写出恰当的子程序。

假定想得到某个范围内的数字（如范围操作符），从小到大，或者从大到小。范围操作符只能从小到大，但可以很容易的弥补这种缺陷：

```
sub list_from_fred_to_barney {
     if($fred < $barney) {
        #Count upwards from $fred to $barney
        $fred .. $banrey
  } else {
    #Count downwards from $fred to $barney
    reverse $barney . . $fred;
    }
    $fred = 11;
    $barney = 6;
    @c = &list_from_fred_to_barney; #@c 为（11，10，9，8，7，6）
```

在本例中，范围操作符给我们 6 到 11 的列表，reverse 倒转此列表，因此得到从 11 到 6 的列表。

也可以什么都不返回。如果 return 后没有任何参数则在标量 context 中将返回 undef，而在列表 context 中将返回空列表。这有助于检验子程序是否返回了正确信息，提示调用者当前不能返回更有意义的值。

4．11 练习

附录 A 中有答案：

1．[12]写一个名为 &total 的子程序，返回一列数字的和。提示：子程序不应当有任何的 I/O 操作；它处理调用的参数，返回处理后的值给调用者。结合下面的程序来练习，它检测此子程序是否正常工作。第一组数组之和我 25。

```
my @fred = qw{ 1 3 5 7 9 };
my $fred_total = &total(@fred);
print "The total of \@fred is $fred_total.\n";
print "Enter some numbers on separate lines: ";
my $user_total = &total(<STDIN>);
print "The total of those numbers is $user_total.\n";
```

2．[5]利用上题的子程序，写一个程序计算从 1 到 1000 的数字的和

3．[18]额外的练习：写一个子程序，名为 &above_average，将一列数字作为其参数，返回所有大于平均值的数字（提示：另外写一个子程序来计算平均值，总和除以数字的个数）。利用下面的程序进行测试：

```
my @fred = &above_average(1..10);
print "\@fred is @fred\n";
print "(Should be 6 7 8 9 10)\n";
my @barney = &above_average(100, 1..10);
print "\@barney is @barney\n";
print "(Should be just 100)\n";
```

第五章 输入与输出

我们在早期练习中已经使用过一些输入/输出(I/O)操作。现在我们将学习更多的此类操作，其涵盖了大概 80%的输入/输出内容。如果熟悉标准输入，输出，错误流，将更有利于本章的学习。如果不熟悉，通过本章的学习你将掌握这类知识。现在，想象“标准输入（standard input）”为“键盘”，“标准输出(standard output)”为“显示器”。

5．1 从标准输入设备输入

从标准输入设备输入是容易的。使用<STDIN>，我们已经见过了◆。在标量 context 中它将返回输入的下一行：

◆这里，我们称：<STDIN>为行输入操作，但其实际上是对一个文件句柄(filehandle)的行输入操作(有<>表示)。本章后面将更多的介绍文件句柄(filehandle)。

```
$line = <STDIN>;                    #读入下一行；
chomp($line);                       #去掉结尾的换行符

chomp($line=<STDIN>)                #同上，更常用的方法
```

由于，行输入操作在到达文件的结尾时将返回 undef，这对于从循环退出时非常方便的：

```
while (defined($line = <STDIN>)) {
    print   “I saw $line”;
}
```

第一行代码值得仔细说明：我们将输入的字符串读入一个变量，检查其是否 defined，如果是（意味着我们没有到达输入的结尾），则执行 while 的循环体。因此，在循环的内部，我们将看到每一行，一行接着一行◆。你可能经常进行此类操作，因此 Perl 提供了一种简写方式。如下：

◆你可能注意到我们没有对输入的字符串进行 chomp 操作。在这种类型的循环中，你不能将 chomp 操作插入条件表达式中，因此这通常是循环体的第一条语句。我们将在下一节中见到此类例子。

```
while(<STDIN>){
  print “I saw $_”;
}
```

为了得到这种简写，Larry 使用了一些无用的语法(useless syntax)。也就是说：“读入一行输入，看其是否为真(通常为真)。如果为真，则进入 while 循环，然后丢掉它！”。Larry 知道很少有人这样做；没人会在实际的程序中使用这种方法。因此，Larry 利用了这种特性，使之变得有用起来。

上段程序实际完成的工作和早期例子是一样的：它告诉 Perl 将输入读入一个变量，（只要结果是 defined 的，即没有到达文件的结尾），则进入 while 循环。但，不是将输入存储在$line，而是放在 Perl 的默认变量,$_之中。你也可以这样写：

```
while (defined($_ = <STDIN>)){
        print "I saw $_";
}
```

在进行深入讨论前，我们要澄清一些事：这种简写只在特定的情况下有效，默认的情况下不会将一行读入变量$_。仅当 while 循环的条件判断部分只包含行输入操作才有效◆。如果在条件判断部分还有别的内容，则上述简写无效。

◆由于 for 循环的条件部分和 while 的条件部分类似，它也能正常工作。

除此之外，行输入操作(<STDIN>)和 Perl 的默认变量（$_）没有别的关系。在本例中，输入的值被保存在变量$_中。

另一方面，在列表 context 中使用行输入操作时，则会将所有的行（剩下的）当作一个列表，而每一行作为列表的一个元素：

```
foreach(<STDIN>){
    print "I saw $_";
}
```

同样，行输入操作和 Perl 的默认变量$_没有必然的联系。在上例中，foreach 默认的控制变量为$_。因此，此循环将每一行赋给$_。

上面的看起来相似：好像有理由认为它们的行为也是相似的，不是吗？

它们在底层是很不相同的。在 while 循环中，Perl 读入一行，将它赋给变量，然后进入循环。再回到开头，读入下一行。但在 foreach 循环中，由于行输入操作在列表的 context 中使用，因为 foreach 需要一个列表作为其参数。因此，它在循环执行前会将所有的输入读入。这种区别在读入一个 400MB 的 web 服务器的 log 文件时非常明显。通常使用 while 循环是一种更好的方法，因为它一次处理一行输入。

5．2 从<>输入

另一种方法是使用尖括号◆输入(diamond operator)：<>。这种方法对于书写类似于标准 Unix◆工具的程序非常有用。如果想写一个 Perl 程序，使它具有像 *cat*, *sed*, *awk*, *sort*, *grep*, *lpr*, 以及许多别的应用程序类似的功能，则<>将帮上你的大忙。对于其它方面，<>可能帮不上你什么。

◆这种操作由 Larry 的女儿，Heidi 命名。有一次 Randal 到 Larry 的家去， 给他看一些他一些新写的培训材料，抱怨到不知道怎么称呼它（<>）。Larry 也不知道怎么叫它。Heidi（当时只有 8 岁）突然说， "That's diamond, Daddy"。因此就这样称呼它了。谢谢，Heidi。

◆不仅仅在 Unix 系统中。许多别的系统也采用了这种 invocation arguments（调用参数）的方法。

一个程序的调用参数（invocation arguments）, 通常是命令行中程序名字后面的一些"字符串"◆。下例中，它们是要被顺

序处理的文件的名字：

◆当程序开始运行运行时，它有 0 个或多个调用参数，这由此程序决定。这通常出现在 shell 中，此列表由你命令行中输入的内容决定。在后面将看到，调用程序可以使用许多字符串作为调用参数（invocation arguments）。由于它们经常出现在命令行中，因此有时亦被称作命令行参数（command-line arguments）。

```
$ ./my_program   fred   barney   betty
```

上述命令的含义是，运行 my_program（在当前目录下），它将处理文件 fred, 再处理文件 barney， 最后是文件 betty。

如果没有命令行参数，程序将处理标准输入流（standard input stream）。作为一个特例，如果将连接号（-）作为一个参数，其含义也是标准输入◆。如果调用参数为 fred – betty, 其含义是程序将首先处理文件 fred，其次是标准输入流，最后是文件 betty。

◆这是 Unix 中很少人知道的一个事实：如许多标准的工具，如 *cat*, *sed*, 也使用这种约定，连接号（-）代表标准输入流。

使用这种方法来书写程序的一个好处是，可以在程序运行时决定其输入。例如，你不需要重写它就可以在管道（pipeline）中使用（将在后面讨论）。Larry 将它引入 Perl， 是因为他希望可以容易的写你自己的程序，使它们具有标准 Unix 工具的特点，甚至是在 Non-Unix 系统中。事实上， Larry 本人就做了大量的此类程序。由于不同工具提供商提供的工具不同，Larry 可以创建他自己的工具，这些工具可以在不同的机器上使用，并且其行为相同的。当然，这意味着要首先将 Perl 移植到这些机器上去。

尖括号操作（<>）是一种特殊的行输入操作。其输入可由用户选择◆：

◆可能是从键盘，或者不是

```
while (defined($line = <>)){
   chomp($line);
   print “It was &line that I saw!\n”;
}
```

运行此程序，调用参数为 fred, barney, betty, 则结果大概如下：“It was [a line from file fred]（文件中 fred 的一行） that I saw!”, “It was [another line from file fred]（文件 fred 中的另一行） that I saw!” , 直到文件 fred 的结尾。然后，将自动转到文件 barney， 一行一行的输出，最后到文件 betty。从一个文件到另一个文件之间没有空行，当使用<>时，就像输入的是一个大文件一样◆。如果输入结束时，<>将返回 undef（同时退出 while 循环）。

◆当前的文件名字被保存在 Perl 的特殊变量$ARGV 中。名字 “-”代表某个文件，如果其为标准输入流输入。

由于这是一种特殊的行输入操作，我们也可以使用前面的相似的简写方法：

```
while(<>){
   chomp;
   print “It was $_ that I saw!\n”;
}
```

这种方法和前一例的功能一样，但输入的字符更少。你可能注意到了，chomp 使用了默认参数，没有变量时，chomp 将对$_操作。这又减少了输入。

由于<>通常被用来处理所有的输入，因此在同一个序中重复使用是不正确的。如果在同一个程序中使用了 2 次<>，特别是在 while 循环内第二次使用<>读入第一次<>的值，其结果通常不是你所希望的◆。根据我们的经验，当初学者在程序中使用第二个<>，通常是想使用$_。记住，<>读入输入，但输入内容本身被存储在$_(默认的情形)。

◆如果在第二次使用<>之前重新初始化@ARGV，则能得到正确的结果。在下一节将介绍@ARGV。

如果<>不能打开文件从中读入，它将打印出一些有用的诊断信息，如：

```
Can't open wilma: no such file or directory
```

<>将自动转到下一个文件，这和 cat 这个标准工具是一致的。

5．3 调用参数

技术上讲，<>从数组@ARGV 中得到调用参数。这个数组是 Perl 中的一个特殊数组，其包含调用参数的列表。 换句话说，这和一般数组没什么两样（除了其名字有些特别：全为大写字母），程序开始运行时，调用参数已被存在@ARGV 之中了◆。

◆C 程序员可能想到 argc（Perl 中没有），和程序自身的名字存放在什么地方等（它被存在 Perl 的特殊变量$0 中，而非@RAGV）. 对于不同的调用方式， 这可能有些不同。具体的情况可参看 perlrun。

可以像数组那样使用@ARGV，如使用 shift 将元素移出，或者使用 foreach 进行迭代等。甚者可以检查是否某个参数由-开头，进而可以做为参数选项(invocation options)处理它们（如 Perl 处理其-w 选项）◆。

◆如果需要更多的这种选项，那很可能你是使用模块按照标准方法处理它们。参看 Getopt::Long 和 Getopt::Std 这两个模块，它们属于标准发布的 Perl 中的一部分。

<>操作查看@argv 来决定使用哪些文件。如果表为空，则使用标准输入流；否则，使用其找到的相应文件。也就是说，在启动程序后，使用<>之前，你还有机会修改@argv 的值。例如，下面程序可以处理 3 个指定的文件，无论用户在命令行中输入了什么其它的文件：

```
@argv = qw# larry mor curly #;                    #强制使用这三个文件
   while(<>){
   chomp;
   print "It was $_ that I saw in some stooge-like file!\n";
}
```

5．4 输出到标准输出设备

print 操作将传给它的依次输出到标准输出设备中，一个接着一个。不会在这些元素之前，之后，或之中加入任何字符◆。如果想在其间加入空格，在结尾加入换行符，那需具体指定它：

◆默认情况下，不会加入任何的字符，但这种默认行为(就像 Perl 中别的 default 一样)是可以改变的。改变这种默认值，可能影响你的维护人员，因此应当尽量避免它，除非在一些小的，quick-and-dirty 程序，或者程序的一小段中。参看 perlvar 的帮助手册，学习改变这种默认值的方法。

```
$name = “Larry Wall”;
print “Hello there, $name , did you know that 3+4 is”, 3+4, “?\n”;
```

当然，打印数组和内插一个数组是不同的：

```
print @array;                    #打印出元素的列表
print “@array”;                #打印一个字符串(包含一个内插的数组)
```

第一个语句打印出所有的元素，一个接着一个，其中没有空格。第二个打印出一个元素，它为@array 的所有元素，其被存在一个字符串中。也就是说，打印出@array 的所有元素，并由空格分开◆。如果@array 包含 qw /fred barney betty /◆，则第一个例子输出为：fredbarneybetty，而第二个例子输出为 fred barney betty（由空格分开）。

◆是的，空格也为默认值。参见 perlval

◆你知道此处我们指的是三个元素的列表，对吧?

在决定使用第二种方法前，想象@array 为一个 unchomped 的输入列。也就是说，每一个字符串都有一个换行符作为后缀。现在第一个 print 语句输出为 fred, barney, betty，分别在三行中。第二个 print 语句输出为:

```
fred
 barney
 betty
```

你知道空格是哪里来的吗?Perl 在内插数组时，它会在元素之间加入空格。我们得到数组的第一个元素(fred 和换行符)，空格，数组的第二个元素(barney 和换行符)，空格，数组的最后一个元素(betty 和换行符)。结果是数组除了第一个元素之外元素都被缩进了。每隔一周或者两周，comp.lang.perl.misc 的新闻组上就会出现如下的消息:

不用阅读这些消息，我们就知道程序中对数组使用了双引号，数组含有 unchomped（没有去掉末尾的换行符：译者注）的字符串。当问到:“你对 unchomped 的数组使用了双引号吗?”，答案通常是 yes.

如果，字符串包含换行符，如果想输出它们，通常:

```
print @array;
```

如果不包含换行符，通常想加上一个:

```
print "@array\n";
```

如果使用了双引号，通常要加上\n。这能帮助你区别开来。

通常，需要把程序的输出先缓存起来。将要输出的内容先缓存起来，等到有足够的内容再输入，而非立刻就输出。例如，假设要把输出的内容存入磁盘，如果有一个或两个字符就立刻输出到磁盘中，这将非常缓慢和低效。一般，先将要输出的内容存入一个缓存(buffer)中，当缓存满时，再将其输出。通常，我们希望这样做。

但如果你的程序需要你立刻输出，则希望每一次 print 时，就输出。参照 Perl 的帮助手册，了解更多的关于控制缓存的信息。

由于 print 需要一些输出的字符串列表，则其参数是作为列表 context 来求值的。由于<>操作返回的是列表，则它们可很好的一起工作：

```
print <>;                    # 'cat'的源程序
print sort <>;             #'sort'的源程序
```

坦白讲，标准 Unix 命令 *cat* 和 *sort* 还有些其它功能是上述程序所缺乏的，但不能低估上述程序的价值！现在你可以使用 Perl 来实现标准的 Unix 工具，再将其移植到任何装有 Perl 的机器上，无论上述机器是否运行 Unix。并且，上述程序在不同的机器上有相同的行为◆。

◆事实上，Perl Power Tools(PPT) 项目的目标是将所有传统的 Unix 工具在 Perl 中实现，它们几乎实现了所有的工具（包括大多数的游戏），当在如何实现 shell 时被难住了。PPT 项目是成功的，因为在大多数 non-UNIX 系统中实现了标准的 Unix 工具。

还有一点需要说明的是，print 还有可选的括号，它在某些时候可能引起混淆。记住 Perl 中关于括号的规则：如果括号不能改变语句的意义，则其就是可选的。下面是输出某个相同语句的不同的方法:

```
print ("Hello, world!\n");
pirnt "Hello, world!\n";
```

Perl 的另一个规则是：如果调用 print 的方法看起来是函数调用，则它就是函数调用。这条规则很简单，但看起来像函数调用是啥含义呢？

在函数调用中，函数名字后面会紧接着◆括号，其中为函数的参数，如下：

◆我们说 "紧接着"，是说 Perl 在这类函数调用中，函数名字和开括号之间不能有换行符。如果有换行符，Perl 则将此代码看作列表操作，而非函数调用。想了解更多的信息，可以参看帮助手册。

```
print (2+3);
```

上述代码看起来像函数调用，它实际上就是函数调用。其输出为 5，然后返回一个值，这和其它函数调用一样。print 返回值为 true 或者 false，表明 print 成功或者失败。通常是成功的，除非遇到 I/O 错误，因此，下面语句中$result 的值通常为 1:

```
$result = print("hello world\n");
```

如果以其它方式来使用返回值呢？假如想把返回值乘以 4：

```
print (2+3)*4;    #oops!
```

当 Perl 遇到上述代码时，将输出 5。然后将 print 的返回值 1，乘以 4。其结果被丢弃了，因为没有将其值赋给任何变量。现在，有些人可能会说："嘿，Perl 不能进行数学运算！应该输出 20，而非 5！"

这是由括号是可选的引起的；有时，人会忽略括号属于哪一部分。当没有括号时，print 是一个列表操作，将后面的所有元素输出来，这通常是你所希望的。当 print 后面是一个开括号时，print 为函数调用，它将输出括号中的内容。由于上述代码中有括号，则它等同于下面的代码：

```
(print (2+3)) * 4;       #oops!
```

幸运的是，如果打开了警告时(warnings)，Perl 会提示你。因此至少在程序的开发和调试阶段，使用 –w 或者 use warnings，。

上述规则，对于 Perl 中所有的列表函数都是有效的，不仅仅是 print，但 print 可能是最容易引起人注意的。如果 print(或者别的函数)后面紧接着一个开括号，则要确保闭括号在所有的参数之后。

◆0 参数或者 1 参数的函数不会遇到这种问题。

5．5 使用 printf 格式化输出

你可能需要对输出有比 print 更多的控制。也许，你习惯了 C 提供的格式化输出函数：printf，担心 Perl 中没有提供相同的名字，类似功能的的函数。

printf 函数有一个格式字符串(format string)，后接需要输出的字符串列表。格式◆字符串，是一个模板，它规定输出的格式：

◆这里，我们按照一般意义来使用 "format"。Perl 有报告生成(report-generating)的功能，称为 "formats"。现在我们不打算讨论它。

```
printf "Hello, %s; your password expires in %d days!\n",
    $user, $days_to_die;
```

格式字符串中有一些被称作格式符(conversion)的东西；每一个格式符由百分号(%)开头，由字母结束。(很快我们将看到，在这两个字符之间可以有其它字符)。后面元素个数和格式符的个数是一致的。如果不等，则不能正确执行。上述代码中后面有 2 个元素，有 2 个格式符。因此输出类似于：

```
Hello, meryn; your password expires in 3 days!
```

printf 有许多的的格式符，此刻我们介绍最常用的。当然，详细的信息请参见 perlfunc 的帮助手册。

要输出数字，通常使用%g◆，它将根据需要自动选用浮点数，整数，或者指数：

◆ “Geneal” numeric conversion（通用数字格式符） 或者是 “Good conversion for this number” （关于这个数字的好的格式符）抑或 “Guess what I want the output to look like.”（猜猜我将怎样输出）

%d 为十进制◆整数，根据需要而截尾：

◆%x 是针对十六进制的，%o 是针对八进制的。

```
print “in %d days!\n”,17.85;    #in 17 days!
```

上述结果被截尾了，而非变成和它最相近的整数；后面将讨论怎样变成最相近的整数。

在 Perl 中，printf 中的数据通常会接受一个宽度值。如果数据不能满足这个宽度，则会自动扩展开来：

```
printf   “%6f\n”, 42;                  #输出为   ○○○○42（○此处指代空格）
printf   “%23\n”,2e3+1.95;              #2001
```

%s 是针对字符串的，例如：

```
printf “%10s\n”, “wilma”;   #输出为：○○○○○wilma
```

如果宽度值为负数，则为左对齐（对于所有的格式符）：

```
print “%-15s\n”, “flintstone”;   #输出为 flintstone○○○○
```

%f 根据需要进行截尾，你可以设置需要几位小数点后面的数字：

```
printf “%12f\n”, 6*7 + 2/3;          #输出为：    ○○○42.666667
printf “%12.3f\n”, 6*7 + 2/3;        #输出为：    ○○○○○○42.667
printf “%12.0f\n”, 6*7 + 2/3;        #输出为：   ○○○○○○○○○○43
```

要输出一个百分号，可以使用%%，它不会使用后面列表中的元素◆：

◆你可能认为可以在百分号前使用反斜线（\）。你可以试试，但这是不正确的。它不能正确工作的理由是，格式是表达式，“\%”是指一个字符的字符串‘%’。故虽然我们在格式字符串中使用了反斜线，printf 函数并不知道怎样处理它。C 程序员可以使用者这种方法来处理。

```
print “Monthly interest rate: %.2f%%\n”,
5.25/12;   #输出的值为  0.44%
```

5.5.1 数组和 printf

通常，不会将数组作为参数传递给 printf。因为数组可能含有任意数量的元素，而某个给定的格式字符串仅对固定数量的参

数有效：如果格式化字符串中有 3 个格式符，则对应 3 个元素。

但没有理由因为表达式可以是任意的，就不能指定一个格式字符串。这需要一些技巧才能使之正常工作，如果将格式存入变量中可能会带来方便（特别是调试的时候）：

```
my @items = qw( wilma dino pebbles );
my $format = "The items are:\n" . ("%10s\n" x @items);
## print "the format is >>$format<<\n";     #用于调试
printf $format, @items;
```

上述代码中使用了 x 操作（在第二章中学习过），来确定字符串要重复的次数，此次数有@items 确定（此时是在标量 context 中使用的）。本例中为 3，因为有 3 个元素，因此上述的格式字符串等同于 "The items are:\n10s\n10s\n10s\n"。输出的每一个元素占一行，右对齐，长度为 10。相当酷，对吧？还有更好的方法，可以把它们结合起来使用：

```
printf "The items are:\n" . ("%10s\n" x @items), @items;
```

本例中使用了@items 两次，一次在标量 context 中，取其元素的个数，一次在列表 context 中取其元素。context 是相当重要的。

5．6 句柄

文件句柄（filehandle）是 Perl 程序 I/O 连接的名字，是 Perl 和外界的纽带。也就是说，它是连接的名字，而非文件的名字。

文件句柄的命名规则和 Perl 中其它标识符一样（由字母，数字，下划线组成，但不能由数字开头）；由于没有任何的前缀符，这可能和现在或者将来的保留字，标签（在第十章中介绍）混淆。因此，和标签一样，Larry 推荐文件句柄的所有字母均大写。这有利于阅读，并且能保证程序在引入新的保留字（小写）后仍能正确执行。

Perl 自身有六个文件句柄：STDIN，STDOUT，STDERR，DATA，ARGV，ARGVOUT◆。虽然可以任意给文件句柄命名，但不能选择上面六个，除非你想利用它们的某些特殊性质◆。

◆有些人不喜欢使用全部大写的字符串，他们喜欢使用小写字母，如 stdin。Perl 可能让它正确执行，但这并非是普适的。它们在何时正确执行，何时会失败不在本书讨论范围之内。需要指出的是，如果你的程序依赖这种性质，即便它现在能正确执行，也很可能在将来的某一天失败，因此最好避免使用小写的形式。

◆某个情况下，这要使用不会有任何问题。当你的维护人员可能会混淆。

你可能还记得其中一些句柄的名字。当程序运行时，STDIN 连接 Perl 当前处理的部分和其他输入来源，其一般被称作标准输入流。通常，是指键盘，除非指定了别的输入源，如文件或者另一个程序的输出，通过管道(pipe)◆。STDOUT 是标准输出流。默认情况，这是指用户的显示屏，但可以将其输出到文件，或者另一个程序中，我们将很快看到。这些标准流来源于 Unix 的标准 I/O 库，但它们在大多数的当代操作系统◆中都能正确工作。一般的思想是，程序只是从 STDIN 读入，写出到 STDOUT，相信用户（或者启动此程序的程序）能正确地设定。由于上述原因，用户就可以在 shell 提示符中输入如下的命令：

◆本章中我们讨论的 3 种主要的 I/O 流是 Unix shell 默认情况下使用的。但不是只有 shell 能调入程序。我们在第十四章中讨论当由 Perl 调入时将发生什么事。

◆如果不熟悉你的 Non-Unix 系统提供的标准输入输出设备，可以参看 perlport 的帮助手册，来查看其和 Unix shell（运行程序的程序，依赖于键盘的输入）的等价物。

```
$ ./your_program <dino   >Wilma
```

上述命令告诉 shell，从一个名叫 dino 读入，将结果输出到叫做 wilma 的文件之中。由于程序从 STDIN 读入，处理它（按照我们的要求），再输出到 STDOUT，上述代码将能很好工作。

不需要额外的代价，上述程序能很好的在管道中使用。这是 Unix 的另一个概念，可以让我们如下的写命令：

```
$ cat fred barney | sort | ./your_program | grep something | lpr
```

如果你不熟悉 Unix 命令，也不要紧。上面一行的含义是：cat 命令将输出文件 fred 的所有行，紧接着是文件 barney 的所有行。这个输出作为 sort 的输入，它将所有的输入的行进行排序，再将结果传递给 your_program。经过 your_program 处理后，将结果传给 grep，它会将某些行去除掉，然后送给 lpr，它会将传给它的数据打印出来。

像上面那样的管道命令在 Unix 和许多别的系统中非常常见，因为它们可以让你通过简单的，标准命令创建强大的，复杂的命令。每一个小的命令完成一件事情，你的任务就是准确地组合它们。

还有一种标准的 I/O 流。如果 your_program 有警告(warnings)或者诊断(diagnostic)信息，这些信息不应当在管道中传递。grep 命令可以去掉任何没有特别指定的信息，因此很可能去掉了 warnings。即便它保留了这些 warnings，你很可能不希望把这些信息在管道中传递下去。这就是为什么还有标准错误流的原因：STDERR。标准输出可以输出到另一个程序或者文件，错误可以输出到用户指定的任何地方。默认情况下，错误将输出到用户的显示屏◆，但用户可能把错误输出到文件中，如下面的命令：

◆通常，错误是不会被缓存的。这就使说，如果标准输出流和标准错误都是输出到同一个地方（如显示器），则错误通常会出现在普通输出的前面。例如，你的程序输出普通文件的一行，和除以 0，则通常会先显示除以 0 的（错误）信息，然后才是文件的内容。

```
$netstat | ./your_program   2>/tmp/my_errors
```

5．7 文件句柄的打开

你已经见过了 Perl 提供的 3 种文件句柄：STDIN, STDOUT, STDERR。当需要其它的文件句柄时，使用 open 操作通知 Perl，Perl 再请求操作系统来建立同外部的连接。下面是一些例子：

```
open CONFIG, “dino”;
open CONFIG, “<dino”;
open BEDROCK, “>fred”;
open LOG, “>>logfile”;
```

第一例中打开了一个名为 CONFIG 的文件句柄，它指向 *dino* 文件。也就是说，文件 *dino* 将被打开，其所包含的数据通过 CONFIG 传给程序。这和从文件中取数据，也可以通过 STDIN 得到是类似的，如果在命令行中使用了 shell 重定位操作如<dino。第二例和第一例类似；它和第一例是一样的，只是 < 明确的指明了 “使用这个文件进行输入操作”，虽然默认的情况就是输入（没有<）◆。

◆这可能引起严重的安全问题。正如我们即将看到的（在第十四章有详细讨论）。文件名可能使用了一些 magical 字符串。如果$name 有用户指定的文件名，打开$name 将允许这些 magical characters 能执行。这对于用户可能非常方便，但其存在安全漏洞. open < $name 将会更加安全，因为它严格指明了将打开的文件名作为输入来使用。但是，这仍不能阻止所有的问题。想了解更多的关于打开文件的信息，特别是和安全相关的信息，可以参看 perlopentut 的帮助手册。

打开文件进行输入，一般不需要使用 <, 我们这里介绍的原因是，在第三例中，使用了大于号(>)来表明是文件的输出。此例中打开文件句柄 BEDROCK，输出到新文件 *fred* 中。和 shell 重定位中使用的大于号一样，我们将输出送到文件 *fred* 中。如果存在这样的文件，则清空它，并将新的数据写入。

第四个例子中使用了两个大于号(>>)(和 shell 一样)，它打开一个文件，数据追加到文件后面。也就是说，如果文件存在，将把新数据添在后面。如果文件不存在，则和大于号（>）一样，创建文件，并把数据写入。这对于日志(log)文件是非常方便的；程序可以每次添加写的日志到 log 文件中。这也是把第四例中的文件句柄叫做 LOG， 文件叫做 *logfile* 的原因。

可以在文件名的地方使用任何的标量表达式，虽然通常你可能不想这样做：

```
my $selected_output = “my_output”;
open LOG, “> $selected_output”;
```

注意大于号后面的空格。Perl 会忽略它◆，但它防止可能出现的异常情况。如，当$selected_output 为 “>passwd”，则它将变成追加。

◆是的，如果文件名前有空格，Perl 将忽略它。查看 perlfunc 和 perlopentut 了解更多的信息。

Perl 的新版本中（从 Perl5.6 开始），open 支持“3 参数”类型：

```
open CONFIG, “<”, “dino”;
open BEDROCK, “>”, $file_name;
open LOG, “>>”, &logfile_name();
```

这种方法的优点是 Perl 不会将模式（第二个参数）和文件的名字（第三个参数）混淆，这能增加安全性◆。但，如果你希望你的程序能和早期的 Perl 兼容（如上传到 CPAN），则尽量避免使用这种方法，或者标明其只对 5.6 版本之后的有效◆。

◆这方面的安全问题是，它能让某些恶意用户将有恶意的字符注入你的程序之中。当学完正则表达式（第七章开始）之后，你可以利用它对用户的输入进行检测。如果你的程序存在潜在的恶意用户，你可以阅读 Alpaca 中关于安全问题的论述，或者查看 perlsec 的帮助文档。

◆例如 use 5.6.

在本章后面我们将介绍如何使用这些文件句柄。

5．7．1 Bad 文件句柄

Perl 自身不能打开文件。和许多其它语言类似，Perl 请求操作系统来打开文件。当然，操作系统可能拒绝打开，如权限问题，不正确的文件名等等。

如果从一个 bad 文件句柄读入（文件句柄没有恰当的打开），会立刻到达文件结尾(end-of-file)。（具体的 I/O 方法在本章后面有介绍，end-of-file 在标量 context 中由 undef 标明，在列表 context 中由空列表(empty list)标明）。如果写到一个 bad 文件句柄，数据会被悄悄地丢掉。

幸运的是，上述可怕的结果可以被避免。首先，可以使用 –w 或者 use warnings，Perl 通常能告诉我们如果使用了一个 bad 文件句柄。如果不使用它，open 也能通过返回的是真是假来告诉我们（成功打开为真，打开失败返回假）。你可以如下的写代码：

```
my $success = open LOG, “>>logifle”; #将返回值保存在$success 中
if(!$success){
 #打开失败时
 . . .
}
```

你可以这样做，但下一节还有别的方法。

5．7．2 关闭文件句柄

当结束一个文件句柄时，你可以如下这样关闭它：

```
close BEDROCK;
```

当关闭文件句柄时，Perl 将告诉操作系统已经结束了对此数据流的操作，因此应当将输出的数据写到磁盘中，可能某人正在等待使用它◆。如果程序重新打开它（也就是说，使用 open 重新打开此文件句柄），或者退出程序◆，Perl 将自动关闭文件句柄。

◆如果对系统的 I/O 有足够的了解，你将知道更多的信息。通常，文件句柄被关闭时，下面就是可能发生的事： 如果还有输入存在于文件中，则被忽略。如果还有数据存在于管道(pipeline)中，则写入程序收到管道关闭的信号。如果输出到文件或管道中，则缓存被清空（数据被传走）。如果文件句柄被锁住，则解锁。参阅系统的 I/O 文档了解更多的信息。

◆无论什么原因，从程序退出将关闭文件句柄，如果 Perl 被中断，则缓存中的数据被清空。也就是说，如果你的程序由于除以 0 等错误而退出，Perl 将继续运行，确保数据能正确地输出。如果 Perl 停止运行（内存不够，或者收到异常的信号(unexpected singal)），则最后的数据可能不能写到磁盘中。但是，这通常不是重要的问题。

由于这些原因，许多 Perl 程序没有考虑 close。但是，如果希望程序更加整洁，则每一个 open 都应当使用一个 close。通常，最好在不使用一个文件句柄时就立刻将它关闭，无论程序是否立即结束◆。

◆关闭文件句柄将清空缓存，并释放文件的锁。由于可能有人会使用某个文件，因此对于运行时间很长的程序，应当在可能的情况下早点关闭句柄。但大量的程序只运行一秒或者两秒，因此是否关闭无关紧要。关闭文件句柄也会释放某些有限的资源。

5．8 严重错误和 die

我们先岔开下话题。本节中我们讨论不是直接和 I/O 相关的问题，而是怎么从程序中比正常情况更早退出的技术。

当 Perl 内部发生了一个严重错误(fatal error)(例如，除数为 0，或者使用了无效的正则表达式，或者调用了一个未声明的函数)，程序将停止运行，并告诉你失败的原因◆。可以利用 die 函数来创建我们自己的严重错误。

◆这是默认的情况。参看第十六章了解更多的信息。

die 函数将打印出你给它的消息（利用标准错误流），并确保程序退出时为非零（nonzero）的退出状态（exit status）。

你可能不知道它，但每一个 Unix（以及许多当代的操作系统）上的程序均有一个退出状态，来表明是否成功。运行程序的程序（如 *make* 工具程序）通过查看退出状态来分析程序的运行状况。退出状态是一个单字节，不能说明什么问题；传统上，0 表示成功，非 0 表示失败。可能 1 是指命令行中命令参数的语法错误，2 指运行错误，3 指没有找到配置文件；具体情况和具体程序相关。但 0 通常是指一切正常。当退出状态表明失败时，像 *make* 一类的程序就停止进行下一步处理。

现在我们重写上一节的例子，如下：

```
if(!open LOG, “>>logfile”){
    die “Cannot create logfile:$!”;
    }
```

如果 open 失败，则 die 将结束程序，并告诉你不能创建 logfile。 但，消息中的$!是指什么呢？它是系统产生的一些可读的信息。通常，当系统拒绝了我们的请求（如打开文件），$!将告诉你原因（可能是 “权限不够(permission denied)”或者 “(文件不存在)file not found”，针对本例）。这个字符串和你在 C 或者类似的语言中通过 perror 得到的是一样的。在 Perl 中，你可以通过$!◆得到。但如果使用 die 来表明错误，但此错误不是系统请求失败引起的，则不要使用$!，因为其包含的信息和实际的问题无关。它所包含的信息，仅对系统请求失败时有效。

◆在某些 non-Unix 系统中，$!可能还有 error number 7 一类的信息，你需要查阅文档才能知道其具体的涵义。在 Windows 或者 VMS 中，变量$^E 还有额外的诊断信息。

die 还会为你做一件事：他会自动将 Perl 程序的名字和行数◆输出在消息的末尾，因此能轻易的辨别出是哪一个 die 引起的错误。上述代码如果含有权限不够(permission denied)的错误，则其消息如下：

◆当读文件是发生了错误，错误信息将包含此文件的“chunk number”(通常是行数)和文件句柄的名字，因为一般它们对跟踪 bug 有用。

```
Cannot create logfile: permission denied at your_program line 1234.
```

它们是有帮助的，能告诉你更多的关于错误信息。如果不想要函数和文件的名字，只需在 die 消息后面加上换行符。下面是另一种使用 die 的方法：

```
if(@ARGV < 2){
  die “Not enough arguments\n”;
}
```

如果命令行中的参数少于 2 个，上述程序将打印出此错误并退出。他不会打印出程序的名字和程序出错的行号，这些信息对用户是没有用的；毕竟这是用户的错误。作为一般规则，如果用法错误则在消息后面加上换行符；如果是其它错误，需要利用它来调试，则不要加上换行符◆。

◆程序的名字在 Perl 的特殊变量$0 中，因此你可能将它包含在：“$0:Not enough arguments\n”。这对于在管道中的程序或者 shell 脚本是非常有用的。因为：$0 在程序运行时不断的变化。你可以查看 _ _FILE_ _和_ _LINE_ _（或者 caller 函数）来了解被加上换行符后忽略的信息，从而可以按你自己的格式输出它们。

你应当检测 open 的返回值，因为程序的剩余部分依赖于是否成功的打开它。

5．8．1 警告信息和 warn

如 die 函数可以指定某个严重错误，就像 Perl 的内嵌错误一样（如除数为 0）。也可以使用 warn 函数来产生警告信息，就像 Perl 内嵌的警告一样（当需要其是 defined 的，但使用了 undef 的值）。

warn 函数像 die 那样工作，除了最后一步，它不会从程序中退出。它也能加上程序的名字和行号，并把消息输出到标准错误那里(standard error)，和 die 一样◆。

◆警告(warnings)不能通过 eval 来捕捉，但严重错误(fatal error)可以。但可以在_ _warning_ _（在 perlvar 的帮助手册中有）中找到。

谈论完 die 和 warning 后，我们继续讨论正常的 I/O。

5．9 使用文件句柄

当某个文件句柄被打开进行输入时，可以像从 STDIN 中输入一样。例如，读入 Unix 中的密码文件：

```
if(! open PASSWD, “/etc/passwd”){
  die “How did you get logged in?($!)”;
  }
  while(<PASSWD>){
    chomp;
    . . .
  }
```

在本例中，die 后面的消息使用了$!。它被括号括起来了。如你所知的，“行输入操作(line-input operator)”由两部分组成；尖括号（<>，真正的行输入操作(line-input operator)）和尖括号中的文件句柄。

写出(>)或追加的(>>)的文件句柄，可以和 print 或 printf 结合使用，如：

```
print   LOG "Captain's log, stardate 3.14159\n";                #输出到 LOG 中
printf   STDERR "%D percent complete.\n", $done/$total * 100;
```

注意到文件句柄和要打印的内容之间没有逗号了吗◆？如果使用括号，看起来会非常古怪。下面的两种写法都是正确的：

◆如果你英语或者语言学方面的成绩是 A，当我们说这是“间接对象语法(indirect object syntax)”，你可能说：“是的，当然我知道文件句柄后面为什么没有括号，因为它是一个间接对象(indirect object)”。我没有得到 A，因此我不知道为什么这里不用逗号，但我们不关心它，因为 Larry 告诉我们这样做。

```
printf (STDERR "%d percent complete.\n", $done/$total * 100);
printf STDERR ("%d percent complete.\n", $done/$total * 100);
```

5．9．1 改变默认的输出句柄

默认情况下，如果不指定文件句柄给 print(或者 printf,这里的内容对两者均适用)，则默认会使用 STDOUT。但这个默认属性，可以通过 select 操作进行更改。如下：

```
select BEDROCK;
print "I hope Mr. Slate doesn't find out about this.\n";
print "Wilma!\n";
```

一旦选择了(select)了某个文件句柄，则它将变成默认值。但这通常是一个坏主意，因为会扰乱程序的剩余部分，因此在完成时应当恢复以前的设置◆。默认情况，输出到文件句柄的内容会被缓存起来。将变量$|设置为 1 ，将会在输出操作结束时会立刻清空文件句柄。如果想确保 logfile 能立刻得到消息，以便能观察程序的运行情况，可以使用下面的程序：

◆在少数情况下，STDOUT 不是默认的文件句柄，可以在 perlfunc 的帮助手册中找到相关的设置和重置信息。虽然已经告诉你了相关的帮助手册，还有一点需要说的是：Perl 有两个内嵌的 select 函数，均可在 perlfunc 的帮助手册中找到。另一个 select 函数有四参数，因此有时被称为“四参数的 select”

```
select LOG;
$| = 1;                              #don't keep LOG entries sitting in the buffer
select STDOUT;
#...time passes, babies learn to work, tectonic plates shift, and then ….
print LOG "This gets written to the LOG at once!\n";
```

5．10 重新打开文件句柄

我们早期提到过，如果重新打开一个文件句柄（例如，打开一个文件句柄 FRED，但之前已经打开了一个名为 FRED 的句柄），前一个句柄将被自动关闭。我们也说过不能重用六种标准的文件句柄，除非想利用其特殊的性质。我们还说过 die 和 warn 产生的信息，以及 Perl 内部产生的提示(complaints)信息将自动传到 STDERR 上。如果把这三条信息结合起来看，你

就知道怎样把错误信息传给文件，而非标准错误流◆：

◆不要轻易这样使用。通常让用户在调入程序时设置而非在代码中写死更好。但是，如果程序由其它程序运行，则使用这种方法将非常方便（例如，web 服务器，或者调度程序如:cron，at）。另一个理由是你的程序可能启动一个新的进程（如 system, exec，在第十四章介绍），你希望它们有不同的 I/O 连接。

```
#将出错信息送到私有错误日志上
 if(! Open STDERR, “>>/home/barney/.error_log”){
   die “Can’t open error log for append: $!”;
 }
```

当重新打开 STDERR 时，Perl 中的任何错误信息将被写入新的文件。但如果执行了 die 语句，并且接收错误消息的文件没有打开，会出现什么情况呢？

答案是，如果系统的三个句柄 STDIN, STDOUT, STDERR 重新打开时没有成功，Perl 会自动使用前一个◆。也就是说，只有 Perl 成功的重新打开新的连接，否则是不会关闭以前的连接。

◆至少，如果没有改变 Perl 变量$^F，这是正确的。如果修改了，结果就不定了。

5．11 练习

附录 A 中有下列习题的答案：

1．[7] 写一个程序，类似于 cat，但保持输出的顺序关系。（某些系统的名字可能是 tac。）如果运行此程序：./tac fred barney betty, 输出将是文件 *betty* 的内容，从最后一行到第一行，然后是 *barney*, 最后是 *fred*, 同样是从最后一行到第一行。（注意使用 ./确保调用的是你自己的程序，而非系统提供的）

2．[8]写一个程序，要求用户在不同的行中输入一些字符串，将此字符串打印出来，规则是：每一条占 20 个字符宽度，右对齐。 为了确保正确的输出，在开头打印出一串数字作为比较（帮助调试）。注意，不要犯 19 个字符宽度的错误。例如，如果输入，hello, good-bye，则输出为：

```
1234567890123456789012345678901234567890123456789012345678901234567890
               hello
            good-bye
```

3．[8]修改上一个程序，允许用户选择宽度，如，用户输入 30，hello, good-bye(在不同的行中)，则每一行的宽度为 30。（提示：参阅第二章相应部分）。提示，如果选择的宽度太长，可以增加比较行的长度。

第六章 哈 希

本章，你将看到使 Perl 成为主流程序语言的功能-哈希(hash）◆。虽然哈希是一种非常有效的功能，但许多语言并没有提供。当你学过哈希之后，几乎在你的每一个程序中都会使用到它， 因为它是如此重要。

◆早期，我们把它叫做 “关联数组(associative arrays)”。Perl 组织认为它难于书写和阅读，因此改名为 “哈希（hashes）”.

6．1 什么是哈希？

哈希是一种数据结构，和数组类似，可以将值存放到其中，或者从中取回值。但是，和数组不同的是，其索引不是数字，而是名字。也就是说，索引（这里，我们将它叫 key）不是数字而是任意的唯一的字符串（参见图 6-1）。

这些 keys 是字符串，因此当从中取值时不是使用如数字 3，而是使用此 hash 元素的名字 wilma。

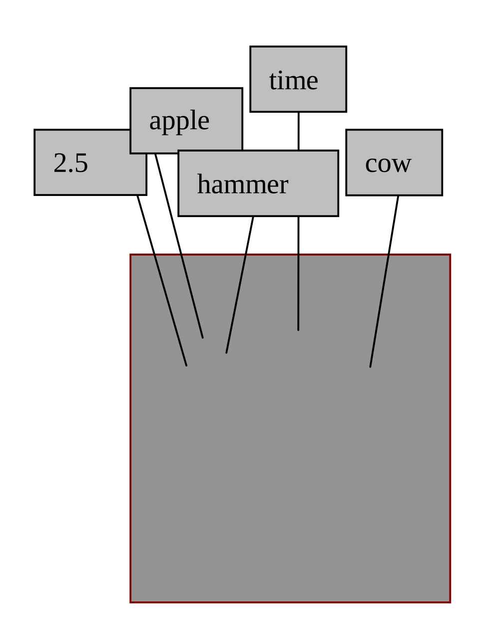
这些 keys 可以是任意的字符串，你可以使用任何的字符串作为 key。但，它们是唯一的；就像数组中只有一个元素的索引是 3，这里也只有一个 hash 元素的名字为 wilma。

另一种思考 hash 的方法是，把它看作一堆数据（a barrel of data）（参见图 6-2），每一个数据都有一个相应的标签。可以通过标签访问此标签对应的元素。但其中是没有 “第一个”元素的概念的。在数组中，数组元素从 0,1,2 开始编号。但在 hash 中，没有确定的顺序，因此也没有第一个元素。只是一些 key/value 对的集合。

图 6-1 哈希的 keys 和 values

keys	values
“foo”	35
“bar”	12.4
“2.5”	“hello”
“Wilma”	1.72e30
“betty”	“bye\n”

图 6-2 一堆数据的 hash



keys 和 values 均为任意的标量，但 keys 通常转换为字符串。因此，如果将表达式 50/20 作为 keys◆，则其通常被转换为 3 字符的字符串 "2.5"。这是上图中的一个 key。

◆此处为数字表达式，而非 5 个字符的字符串"50/20"。如果我们将上述 5 个字符的字符串作为 hash 的 key，则其不会被转变。

通常，由于 Perl 的"没有不必要的限制"的设计哲学：hash 可以是任意大小，从空 hash(没有 key/value 对)，到任何你内存允许的大小。

某些实现了 hash 的语言（如早期的 *awk* 语言，Larry 从中借鉴了 hash），当 hashes 变大时，速度会变慢。Perl 中不会有这种问题，因为它有一个优秀的算法◆。因此，如果 Perl 只有 3 个 key/value 对，它的速度很快。当拥有 3 百万 key/value 对时，其速度仍非常快。大的 hash 表，对性能不会有什么影响。

◆技术上讲，当 hashes 很大时，Perl 会根据需要而重建 hash。"hashes"这个名词的来源即是由于使用了 hash 表来实现它。

keys 是唯一的，但 values 可以重复。hash 的 value 可以是数字，字符串，undef，或者它们的混合◆，但 key 是唯一的。

◆事实上，任意的标量值，包括我们没有在本书见过的标量类型。

6．1．1 为什么使用 Hash?

当第一次听说 hashes，特别是如果以前使用过某种不含 hash 的程序语言仍然高产时，你可能猜想为什么有人要发明这种奇怪的东西呢？通常的理由是，有一类问题是：某个数据集和别的数据集的关系的问题。例如，下面是一些例子：

Given name, family name

Given name（名）是 key，*family name*（姓）为 value。这需要 *given names*(名)是唯一的，如果有两人均叫做 *randal*，这将引起问题。使用 hash，你可以通过查看任何人的 *given name*，而找到其对应的 *family name*。如果使用 key：*tom*， 将得到 value：*phoenix*。

Hostname, IP address

你可能已经知道，因特网上每一台主机都有一个主机名（如 www.stonehence.com）和一个 IP 地址（如 123.45.67.89），因为机器更擅长处理数字，但对人而言，名字更容易记忆。主机名是唯一的，因此可以将其作为 key。在这种 hash 中，我们可以通过查询主机名而找到其对应的 IP 地址。

IP address, hostname

你也可以按相反的顺序来处理。通常我们把 IP 地址看作数字，它也是唯一的字符串，因此也可以作为 hash 的 key。在这种 hash 中，我们可以通过查询 IP 地址来查找对应的主机名。这和上一例中的 hash 是不同的：hash 是单向的，由 key 到 value；没有通过查询 value 来得到其 key 方法。因此，上述 hashes 可以看作一对 hash，一个针对 *IP addresses*(IP 地址)，一个针对 *hostnames*(主机名)。从一个 hash 表得到其对应的 hash 表是很容易的，这在后面将了解到。

Word, count of number of times that word appears(此词出现的次数)

这是 hash 的一种常用用法。它使用的如此普遍，因此，我们在本章中有一个相应的练习。

这里的想法是，你可能希望知道某个文件中单词出现的次数。也许在给一些文档编索引，当用户查询 fred 时，将知道某个文件提到 fred 5 次，另一份文档提到 fred 7 次，第三份文档没有出现 fred 等等。索引的结果将告诉用户那一份文档是其所希望的。当编制此索引的程序读入某个文档时，每当遇到 fred，此文档对应 fred 的 value 就增 1。也就是说，如果某份文档中 fred 已经出现了 2 次，其 value 为 2，当又发现时 fred，其 value 增到 3。如果以前没有找到 fred，则其 value 从 undef(默认值)变为 1。

Username, number of disk blockes they are using[wasting](其使用（耗用）的磁盘块数)

系统管理员喜欢这种应用。某个系统中的 *usernames*（用户名）是唯一的，因此可以把它们作为 hash 的 keys，我们可以查询它了解到此用户的信息。

Driver's license number, name

可能有许多人叫做 John Smith，但每一个人有一个不同的*驾驶证编号(driver's license number)*。这个编号是唯一的，可以作为 key，对应的名字为 value。

还可以把 hash 看作一个简单的数据库，其中每一个 key 下面可以有一块数据。如果你的任务是关于：“查询重复的”，“唯一的”，“交叉引用的”，“查询表”，则 hash 很可能在这类应用中帮上你的忙。

6．2 哈希元素的存取

要访问 hash 元素，可以使用下面的语法：

```
$hash{$some_key}
```

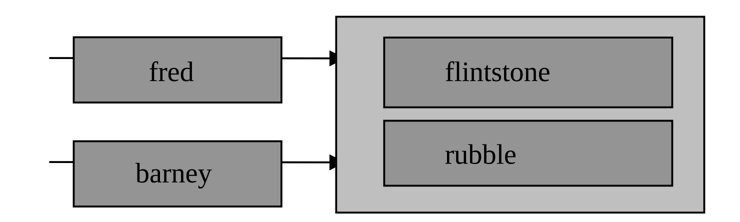
这和访问数组元素的方法有些类似，这里下标(key)上使用的是花括号({})，而不是方括号（[ ]）◆。现在 key 的表达式是字符串，而非数字：

◆这里我们可以窥探到 Larry Wall 的设计思想：Larry 认为由于其和普通数组不同，则也应当使用和普通数组不同的符号。

```
$family_name{“fred”} = “flintstone”;
$family_name{“barney”} = “rubble”;
```

图 6-3 显示了和上例对应的图表。

图 6-3 给 hash 的 keys 赋值



我们如下的书写代码：

```
foreach $person (qw<barney fred>){
        print “I’ve heard of $person $family_name{$person}.\n”;
}
```

hash 的名字和 Perl 中其它的标识符的命名规则是一样的（字母，数字，下划线组成，但不能由数字开头）。由于其属于不同的名字空间，则像 hash 元素$family_name{“fred”}和子程序&family_name 之间没有任何的关系。当然，没有理由将它们命名成一样的，这会让读者困惑。但 Perl 本身并不关心，你是否将某个变量叫做$family_name,某个数组元素称为$family_name[5]等。但人们通常不习惯，通过名字前面或者后面的符号来判断其具体的含义，虽然 Perl 是这样做的。当在名字前面是美元符号($)，后面是花括号({})，则其为 hash 元素。

当给 hash 选择名字时，最好这样思考：hash 元素的名字和 key 之间可以用 for 来连接。如，“the family_name for fred is flintstone(fred 的姓是 flintstone)”。因此，hash 名为 family_name，现在 keys 和 values 之间的关系就相当清楚了。

Hash 的 key 可以是任意的表达式：

```
$foo = “bar”;
```

```
print $family_name{$foo . “ney”};     #输出 “rubble”
```

当将某个值存储在已经存在的 hash 元素中，以前的值会被覆盖：

```
$family_name{“fred”} = “astaire”;              #将新值赋给已经存在的元素
$bedrock = $family_name{“fred”};               #得到 “astaire”；以前的值丢失了
```

这和数组或标量变量中的情况是一样的。如果存放一个新值到$pebbles[17]或$dino 中，则以前的值被替换。如果将新值存在$family_name{“fred”}中，则以前的值同样被替换掉。

可以通过赋值语句对 hash 元素赋值：

```
$family_name{“wilma”} = “flintstone”;                 #新增一个 key(也包括 value)
$family_name{“betty”} .= $family_name{“barney”};     #创建一个新元素
```

这和数组或标量中的情形是一样的：如果以前没有$pebbles[17]和$dino，赋值后就有了。如果以前没有$family_name{“betty”}，则情况是一样的。

访问不存在的 hash 元素得到 undef：

```
$grantie = $family_name{“larry”};   #没有 larry：得到 undef
```

这同样和数组或标量中的情形类似；如果$pebbles[17]和$dino 中没有值，则访问它们得到的结果为 undef。如果$family_name{“larry”}中没有存放值，则其返回 undef。

6．2．1 作为整体的 hash

要引用整个 hash，使用百分号(“%”)作为前缀。前面几页中使用的 hash 的名字为%family_name。

为了方便，hash 可以转换为列表，或者反过来。给 hash 赋值（本例中，参见图 6-1）其类型属于列表 context 赋值，其中列表是 key/value 对◆：

◆虽然可以使用任何列表表达式，但其元素个数必须是偶数，因为由 key/value 对组成。元素个数为奇数是不可靠的，这通常会引起警告。

```
%some_hash = (“foo”, 35, “bar”, 12.4, 2.5, “hello”, “wilma”, 1.72e30, “betty”, “bye\n”);
```

hash 的值（在列表 context 中）是一个 key/value 对的列表：

```
@array_array = %some_hash;
```

我们把这称为：将 hash 展开，并将其 key/value 对返回到一个列表中。返回的 key/value 顺序和存放的顺序可能不同：

```
print “@any_array\n”;
```

#可能得到如下的结果：betty bye(换行)，wilma 1.72e+30 foo 35 2.5 hello bar 12.4

上述顺序看起来很混乱，但对 Perl 来讲很方便，这种顺序 Perl 查询起来特别快。一般使用 hash 时并不关心其实际的顺序，或者有某种简单的方法将它们按要求的顺序排序。

虽然 key/value 对的顺序是杂乱的，但返回的列表中每一个 key 和其 value 是连在一起的。因此，虽然我们不知道 key：foo 会出现在什么地方，但我们知道其 value：35 会紧随其后。

6．2．2 Hash 赋值

很少这样做，但可以使用如下的语法在 hash 之间拷贝：

```
%new_hash = %old_hash;
```

这种运算，Perl 实际做的运算比想象的要多。不像 Pascal 或者 C 语言，这种操作通常是拷贝一块内存。Perl 的数据结构更加复杂。这段代码告诉 Perl 将%old_hash 展开成 key/value 的列表，再将其赋给%new_hash，其将 key/value 对一个一个加入的。

将 hash 转变成其它形式更加常见。例如，我们可以将 hash 反转：

```
%inverse_hash = reverse %any_hash;
```

这会将%any_hash 展开成 key/value 对的列表，其列表如（*key,value,key,value,key,value,*……）。然后，将其反转，得到（*value,key,value,key,value,key,*……）。value 和 key 的位置交换了。当将其存放在%inverse_hash，我们就可以查询对于%any_hash 来说是 value 的字符串，因为现在它对于%inverse_hash 来讲是 key 了。查询到的 value 对于%any_hash 来讲属于 key。这给我们提供了一种方法：查询 “value”(现在成了 key)，得到 “key”(现在成了 value)。

你可能猜测（按照科学的方法，如果你足够聪明）这仅当值也是唯一的情况才能正常工作。否则，我们将得到重复的 key，但 key 必须是唯一的。这里有一条规则：最后一次赋值获胜。也就是说，列表中后面的元素将覆盖掉以前的元素（相同 key 时）。由于我们并不知道 key/value 对的顺序，因此也不能判断哪一对将最终获胜（被使用）。如果想使用这种技术，那你必须确保之前的值是唯一的◆。如我们在早期的 *IP address* 和 *hostname* 的例子中就可以使用：

◆或者你并不关心重复的情况。例如，反转%family_name(keys 为 given name（名）, 而 value 为 family names (姓)) 将很容易判断是否存在此 family name。例如，在反转的 hash 中，如果没有 key: slate，那我们就能确定在最初的 hash 中没有 family name：slate。

```
%ip_address = reverse %host_name;
```

我们可以查询 IP 地址，或者主机名来得到其对应的主机名或者 IP 地址。

6．2．3 大箭头符号（=>）

当给 hash 赋值时，有时并不明显哪些元素是 keys，那些是 values。例如，在下面的赋值中（我们在前面已经见过了），我们需要仔细的计数，“*key,value,key,value,…*”,来判断 2.5 是 key 还是 vlaue：

```
%some_hash = ( “foo”, 35, “bar”, 12.4 ,2.5, “hello”, “Wilma”, 1.72e30, “betty”, “bye\n”);
```

如果 Perl 能提供一种方法，让我们轻易的辨别出哪一个是 key，哪一个是 value，那该有多好？Larry 也想过这个问题，因此发明了大箭头符号(=>) ◆。对于 Perl 来讲，其作用和和逗号（，）类似，因此有时称作 “胖逗号(fat comma)”。Perl 语法中，在需要逗号（，）的时候，都可以使用大箭头符号替换；对于 Perl 来讲，它们是一样的◆。下面是给 hash 赋值的另一种方法：

◆是的，还有小箭头（−>）。它和引用一起使用，这是高级话题。如果你准备好了，可以参见 perlreftut 和 perlref 的帮助手册

◆它们在技术上还是有一点不同：任何大箭头符号(=>) 左侧的 bareword（由字母，数字，下划线，但不是由数字开头，前面有可选的加号或减号，组成的序列）都暗含着由引号括起来了的。因此可以省略掉大箭头符号(=>)左侧 bareword 上的引号。你也可以忽略掉 hash 的花括号中的引号，如果里面只有作为 key 的 bareword.

```
my %last_name = (
    “fred”   =>   “flintstone”,
    “dino”   =>   undef,
    “barney”=> “rubble”;
    “betty” => “rubble”,
);
```

上面代码中，很容易辨别出哪一个是 key，哪一个是 value。注意列表中最后一个逗号。我们早期讨论过，这个逗号是没什么用的，但有时能给我们带来方便；如果我们要加入新的元素到 hash 中，我们只需知道每一行都有 key/value 对，结尾有逗号。Perl 会查看不同元素之间的逗号，以及列表结尾处的逗号（此逗号非必需的）。

6．3 哈希函数

某些有用的函数可以对整个 hash 进行操作。

6．3．1 keys 和 values 函数

keys 函数会返回此 hash 的所有 keys，values 函数将返回所有的 values。如果 hash 中没有元素，则此函数将返回空列表：

```
my %hash = (“a” =>1, “b” =>2, “c” =>3);
my @k = keys %hash;
my @v = values %hash;
```

现在，@k 含有 “a”, “b”, “c”,而@v 含有 1, 2, 3，其顺序可能不同。 记住，Perl 并不维护 hash 中元素的顺序。但，其中 keys 按照某种顺序存储，则其对应的 values 也是这个顺序：如果“b”排在 keys 的最后，则 2 也排在 values 的最后。如果 “c”是第一个，则 3 也是第一个。这种论断是正确的，如果没有对 keys 或 values 做什么修改。如果添加了新元素到 hash 中，Perl 会重新对它们进行排序，以便能更快速的访问它们。 在标量 context 中，这些函数返回 hash 中元素的个数(key/value)。此类操作不需要访问 hash 的每一个元素，其效率是很高的：

```
my $count = keys %hash;   #得到 3，是指有 3 个 key/value 对
```

有时，你可能看到有人将 hash 作为 boolean(true/false)表达式来使用：

```
if(%hash){
    print “That was a true value!\n”;
}
```

上述语句为 true，当且仅当其至少含有一个 key/value 对◆。一般，上述语句是说，“如果 hash 非空…”，但实际中很少这样做。

◆其返回值为一个字符串，对于调试的用户有帮助。其看起来像 “4/16”，但 true 或者 false 是由 hash 的列表是否为空决定的。

6．3．2 each 函数

如果想迭代 hash 的每一个元素（如，检查每一个元素），一种通常的方法是使用 each 函数，它将返回 key/value 对的 2 元素列表◆。当对同一个 hash 函数进行一次迭代时，将返回下一个 key/value 对，直到所有的元素均被访问。如果没有更多的 key/value 对，则 each 函数将返回空表。

◆另一种方法是使用 foreach，这将在本节末尾介绍。

实践中，一般只在 while 循环中使用 each：

```
while (($key, $value) = each %hash){
    print “$key => $value\n”;
}
```

上述代码中，首先，each %hash 将从 hash 中返回一个 key/value 对；假设 key 为 “c”，value 为 3，则列表为 (“c”,3)。这个列表赋给($key, $value)，现在$key 为 “c”，而$value 为 3 。

但上述赋值语句是作为 while 循环的条件表达式，这是在标量 context 中。(更确切的说，这是 boolean context，其希望得到 true/false 值，但 boolean context 是一种特殊的标量 context)。列表赋值语句在标量 context 中返回的是元素的个数，在本例中，是 2 。由于 2 是一个 true 值，我们进入循环，然后输出 c=>3。

第二次进入循环时，each%hash 得到一个新的 key/value 对，例如，返回(“a”,1)。(Perl 会返回一个不同的 key/value 对，因为 Perl 能对其追踪。按照行话来讲，每一个 hash 都有一个 iterator（迭代器）。◆）这两元素被存放在($key, $value)中。由于元

素个数也为 2 ，为真值，则进入 while 循环，输出 a => 1。

◆由于每一个 hash 都有一个私有的迭代器（iterator），因此，使用 each 的循环是可以嵌套的，因为不同的 hash 有不同的迭代器（iterator）。虽然这只是脚注，但我们也应当告诉你，你可以通过使用 keys 或 values 函数，重置迭代器（iterator）。如果新列表加入到此 hash 中，或者 each 函数迭代到最后一个元素，也会重置迭代器（iterator）。另一方面，如果在迭代时加入新的 key/value 对，通常是一个坏主意，这不会重置迭代器（iterator）。但很可能混淆你，以及维护人员。

再进行一次循环，得到结果 b=>2。

现在没有剩余的元素了。当 Perl 执行 each %hash，由于没有 key/value 对，因此返回空列表。空列表赋给($key, $value)，$key 得到 undef, $value 仍然为 undef。

◆由于是在列表 context 中，其返回的不是表明失败的 undef，因为 undef 是一个元素的列表(undef)而零元素非空列表( )。

我们并不需要关心上面的过程，因为它是作为一个整体在 while 循环的条件部分被求值的。列表在一个标量 context 中将返回其元素的个数；在本例中，为 0 。由于 0 时 false 值，则 while 循环结束，执行程序的剩余部分。

当然，each 返回的 key/vlaue 对，顺序是混乱的（它其顺序和 keys 和 values 函数返回的顺序相同）。如果想将其按序排放，可以对它们排序（使用 sort），大致如下：

```
foreach $key (sort keys %hash){
   $value =$hass{$key};
   print "$key => $value\n";
   #也可以不使用额外的临时变量$value
   #print "$key => $hash{key}\n";
}
```

在第十三章中有更多的关于对 hashes 进行排序的介绍。

6．4 哈希的通常用法

到现在为止，一个具体的例子更能把问题说清楚。

假设一个叫做 Bedrock 的图书馆使用 Perl 程序，其中有一个 hash 对借阅者借的书的数量进行跟踪：

```
$books{"fred"} = 3;
$books{"wilma"} = 1;
```

很容易知道 hash 的某个元素是 true 还是 false，像下面这样：

```
if($books{$someone}){
   print "$someone has at least one book checked out.\n";
   }
```

hash 中某些元素为 false：

```
$books{“barney”} = 0;            #没有借书
$books{“pebbles”} = undef;       #从没有借过书
```

由于 Pebbles 从没有结果书，因此其值为 undef，而非 0。
对于每个拥有借书卡(library card)的人都有一个 key。对于这个 key，其对应的 value 是其借书的数量，如果从没有借过，其值为 undef。

6．4．1 exists 函数

要查看 hash 中是否存在某个 key（某人是否有借书卡），可以使用 *exists* 函数，如果 hash 中存在此 key，则返回 true，这和是否有对应的 value 无关：

```
if(exists $books{$dino}){
   print “Hey, there’s a libaray card for dino!\n”;
}
```

这即是说，当且仅当 hash 中（keys % books）的 keys 的列表中存在 dino 时，exists $books{“dino”}才返回 true。

6．4．2 delete 函数

delete 函数将某个给定的 key（包括其对应的 value）从 hash 中删除。(如果不存在这个 key,则什么也不做；不会有警告或者错误信息。)

```
my $person = “betty”;
delete $books{$person}; #将$person 的借书卡删除掉
```

这和 hash 中存储的为 undef 是不同的。使用 exists($books{“betty”})将给出相反的结果。使用 delete 后，hash 中将不会存在此 key；如果其值是 udnef，则 key 是存在的。

本例中，delete 和存储的值为 undef 的不同点在于，前者是将 Betty 的借书卡取走(删除)，而后者是给她一张从没有用过的借书卡。

6．4．3 hash 元素的内插

你可以在双引号的字符串中使用单个 hash 元素：

```
foreach $person (sort keys %books){
```

```
    if($books{$person}){

        print "$person has $books{$person} items\n"    #fred 有 3 个
    }
}
```

但不支持整个 hash 的内插；"%books"仅是六个不同的字符%books◆。你已经见过所有的在双引号中会被内插的特殊字符：$和@，可以使用反斜线(\)得到这样的字符，否则其会被内插。其它字符都不会别内插，会被照常输出◆。

◆如果将整个 hash 按照 key/value 对输出出来，这没有什么用处。你在上章中已经见过了，百分号（%）在 printf 中经常作为格式输出的符号。如果再赋予其它的含义，将带来极大的不便。

◆注意双引号中撇号(') ,左中括号([),小箭头(->),双冒号(::)紧跟在变量名后的情形，他们实际含义可能和你希望的不同。

6．5 练习

答案参见附录 A：

1．[7]写一个程序，提示用户输入 given name（名），并给出其对应的 family name（姓）。使用你知道的人名，或者表 6-1（如果你在计算机上花了太多时间，以致什么人都不认识）：

表 6-1 样本数据

输入	输出
fred	flintstone
barney	rubble
wilma	flintstone

2．[15]写一个程序，读入一串单词(一个单词一行)◆，输出每一个单词出现的次数。(提示：如果某个作为数字使用值是 undefined 的，会自动将它转换为 0。）如果输入单词为 fred，barney，dino，wilma，fred（在不同行中），则输出的 fred 将为 3 。作为额外的练习，可以将输出的单词按照 ASCII 排序。

◆一行一个单词的原因是，我们还没有教你一行中析取不同单词的方法。

第七章 正则表达式

Perl 有许多区别于其它语言的特性。在所有这些特性中，最重要的一条是对正则表达式的支持。它允许你方便，快捷的处理字符串相关的问题。

但是获得这种能力是需要付出代价的。正则表达式(regular expressions)是一种特殊语言写成的程序，内嵌于 Perl 之中（是的，你要学习一门新的程序语言◆。庆幸的是，它非常简单。)。从本章开始，我们将进入正则表达式的世界，现在你可以暂时忘掉 Perl。下一章中，我们将介绍如何在 Perl 中使用正则表达式。

◆ 一些人可能争论说正则表达式不是完备的程序语言。我们不准备在这里争论这个问题。

正则表达式不仅仅是 Perl 的一部分；在 *sed, awk, procmail, grep,* 以及许多程序员文本编辑器(programmer's text editor)如 *vi*, *emacs*，还有一些更深奥的程序中都有它的踪影。注意观察，你可以发现许多工具都支持正则表达式，如 Web 上的搜索引擎（通常由 Perl 书写)，email 客户端，等等。不幸的消息是，不同的正则表达式的实现中，语法会有些微的不同，因此需要学习其不同的地方，例如需要还是不需要反斜线(\)。

7．1 什么是正则表达式?

正则表达式，在 Perl 中通常被称为模式(pattern)：某个模板是否匹配某个字符串◆。由于存在无限的字符串，某个给定的模式将这些字符串分成两类：一类是能匹配的，一类是不能匹配的。这里没有，或者，大概，几乎那样的匹配：要么匹配，要么不匹配。

◆某些学究可能认为这个定义不够严格。他们可能会说 Perl 的模式根本不是真正意义上的正则表达式。如果你想知道更多的关于正则表达式的信息可以参看 Jeffrey Friedl(O'Reily)的书籍《掌握正则表达式》(Mastering Regular Expessions)。

一个模式可以匹配多个字符串：1 个，2 个，3 个，上百个，或者无限个。也可能匹配除了 1 个，多个，或者无限个字符串之外的所有字符串◆。我们将正则表达式看作一种由简单语言实现的程序，这种语言只有一个任务：查找某个字符串，返回"匹配上(it matches)"或者"不匹配(it doesnot match)"◆。这就是它完成的所有工作。

◆将学习到，你可以让某个模式永远匹配或者永不匹配。在极少情况下，这也是有用的，但通常是错误。

◆程序也返回一些信息给 Perl。其中的一种信息是"正则表达式内存(regular expressions memories)"，这将在后面会学习到。

你可能在使用 Unix 的 grep 命令时，见过正则表达式，它会输出匹配上模式的行。例如，如果想查看某个给定文件中是否某行提到过 flint，同时同一行后面提到 stone，那你可以如下的使用 grep 命令：

```
$grep 'flint.*stone' chapter*.txt
chapter3.txt:a piece of flint, a stone which may be used to start a fire by striking
```

chapter3.txt:found obsidian, flint, granite, and small stones of basaltic rock, which
chapter9.txt:a flintlock rifle in poor condition. The sandstone mantle held several

不要将正则表达式和 shell 中的文件名匹配模式，*globs* 混淆了。通常 glob 是指，在 Unix shell 下输入*.pm 将匹配所有结尾为.pm 的文件名。上一个例子使用的 glob 为 chapter*.txt。（你可能已经注意到我们将模式括起来了，来防止 shell 将其作为 glob 来处理）虽然 glob 和正则表达式有许多相同的符号，但其作用并不相同◆。第十二章中将介绍 *globs*。

◆*globs* 有时也被称作模式。但严重的问题是，某些面向初级用户的书籍（可能是菜鸟写得）将 *globs* 叫做“正则表达式”，这绝对是错误的。

7．2 使用简单的模式

要匹配某个模式（正则表达式）和$_的关系，可以将模式放在正斜线(//)之间，如下：

```
$_ = “yabba dabba doo”;
 if(/abba/){
 print “It matched!\n”;
}
```

表达式/abba / 将在$_寻找这四个字母。如果找到，则返回 true，在本例中，它出现了不止一次，但结果没什么不同。总之，如果找到了，则匹配上；如果没找到，则没匹配上。

由于模式匹配通常返回 true 或 false，因此经常用在 if 或 while 的条件表达式部分。

所有在双引号中的转义字符在模式中均有效，因此你可以使用/coke\tsprite/来匹配 11 个字符的字符串 coke, tab(制表符), sprite。

7．2．1 元字符

如果模式只能匹配字面上的字符串，则其用处不会太大。这也是引入特殊字符的原因，它们被叫做元字符(metacharacters)，在正则表达式中具有特殊的含义。

例如，点(.)是通配符，它可以匹配任何单个的字符，但不包括换行符(“\n”)。因此，模式/bet.y/将匹配 betty。同时也匹配 betsy, bet=y, bet.y，或者说任意字符串后接 bet,然后是任意的单个字符（不包括换行符），后接 y。它不会匹配 bety，betsey，因为 t 和 y 之间不是一个字符。点(.)只匹配一个字符。

如果想匹配句号（英语中句号就是一个点：译者注），可以使用点(.)。 但由于点(.)可以匹配任意的单个字符（除换行符外），则其结果比你希望的要多。如果只希望点(.)匹配句号，可以使用反斜线。这条规则对 Perl 正则表达式中所有元字符均有效：元字符前使用反斜线将使它变成普通的字符。如，模式/3\.14159/中的点(.)即不是通配符。

反斜线是第二个元字符。如果需要真正的反斜线，需要重复使用两个反斜线，这和 Perl 中其它情况下是一样的。

7．2．2 简单的量词

通常，需要模式中某些串是可以重复的。星号(*)表示匹配前一项 0 次或者多次。因此，/fred\t*barney/将匹配上 fred 和 barney 之间有任意个制表位(tab)的字符串。它可以匹配“fred\tbarney”，其间有一个 tab；匹配“fred\t\tbarney”，其间有两个制表位；“fred\t\t\tbarney”其间有三个制表位；“fredbarney”，其间什么也没有。这是由于星号(*)是指“0 个或者多个”，因此其间可以是任意个制表符，但不能是其它的字符。可以这样看待星号(*)：“前面的东西，重复任意次数，包括 0 次”（因为*号在数学上是乘法运算符）。

如果希望包括不同的字符，怎么办呢？点(.)可以匹配任何单字符◆，因此.*将匹配任意字符任意多数。这就是说模式/fred.*barney/将匹配 fred,和 barney 之间有任意多个任意字符(不含换行符)的字符串。任意行如果前面有 fred，后面有 barney，其间为任意字符（字符串）都将匹配上。我们将.*叫做“任意字符串匹配模式”，因为任意的字符串均能被匹配上（不包括换行符）。

◆除了换行符。以后我们将不再提醒你，因为你已经知道了。许多时候你不用担心这个问题，因为通常你的字符串中并不含有换行符。但不应该把这个细节忘掉，因为说不定某人就在你的字符串中加入了换行符，因此应当记住点(.)不会匹配换行符。

星号的正是叫法是数量词(quantifier)，意指其可以指代多个前面的项。它不是唯一的数量词，加(+)也是。加(+)的意思是可以匹配前面一项的一个或多个：/fred +barney/意思是 fred 和 barney 之间由空格分开，且只能是空格。(空格不是元字符)。它不会匹配 fredbarney，因为加(+)意指一个或多个，因此至少是一个。可以这样看待加(+)：“最后一项，（可选的）至少还有一项。”

还有第三个数量词，其限制性更强。它是问号(?)，其含义是前面一个项出现一次，或者不出现。也就是说，前面这个项出现 1 次或者 0 次，此外不会有其它情况。因此，/barm-?bamm/只匹配：bamm-bamm 或 bammbamm。这很容易记住：“前面的这个项，出现？或者不出现？”

这三个数量词必须紧跟在某些项的后面，因为它是指前面项重复的次数。

7．2．3 模式中的分组

括号也是元字符。在数学中，括号(())用来表示分组。例如，模式/fred+/能匹配上如 fredddddddd，这样的字符串，但这种字符串在实际中没有什么用途。模式/(fred)+/能匹配上像 fredfredfred 这样的字符串，这更可能是你所希望的。那么模式/(fred)*/呢?它将匹配上像 hello,world 这样的字符串◆。

◆星号 (*) 意指匹配上 0 次或者多次 fred。当为 0 时，那什么字符串都能被匹配上。这个模式能匹配上任何字符串，甚至是空串。

7．2．4 选择符

竖线(|)，在这种用法中通常被读作“或（or）”，意思是匹配左边的或者右边的。如果竖线左边没有匹配上，则匹配右边。因此，/fred|barney|betty/将匹配出现过 fred，或者 barney，或者 betty 的字符串。

现在你可以书写像/fred(|\t)+barney/这样的模式，它将匹配 fred，barney 以及中间由空格，制表符(tab)，或者二者混合所组成的字符串。加(+)是指重复 1 次或多次；每重复一次，(|\t)则有可能匹配一个空格，或者一个制表符◆。但 fred 和 barney 之间这些字符中（空格，制表符）的其中之一必须出现一次。

◆如果使用字符类(character class)进行这种匹配将更有效，这在本章后面会介绍。

如果希望 fred 和 barney 之间的字符是一样的，可以将模式写成/fred(+|\t+)barney/。在本例中，分隔符必须全是空格或者全是制表符。

模式/fred (and|or) barney/能匹配如下两种字符串：fred and barney, fred or barney◆。也可以将模式写成：/fred and barney|fred or barney/，但这样书写的字符更多。并且其效率也更低，这依赖于正则表达式引擎中所使用的优化方法。

◆单词 and 和 or 在正则表达式中不是操作符！它们在正则表达式中就是其本来的含义：单词 and, or。

7．3 字符类

字符类，是方括号[]中的一列字符，可以匹配上括号内出现的任意单个字符。它匹配一个字符，但这个字符可以是列中的任意一个。

例如，字符类[abcwxyz]可以匹配上括号内七个字母中的任意一个。为了方便，我们可以使用连字号(-)来表示某个范围的字母，因此上例也可以写做[a-cw-z]。上面例子省略的字符不多，但像[a-zA-Z]将非常方便，利用它不需要输入 52 个字符◆。你可以使用和双引号相同的字符简写方法，例如类[\000-\177]可以匹配上任意的七比特的 ASCII 字符。◆。当然，字符类只是模式的一部分，单独的字符类在 Perl 中没什么实际的意义。例如，你可能见到如下的代码：

◆注意这 52 个字母不包括 Å, É, Î, Ø, 和 Ü。如果允许处理 Unicode，则上述字符的范围将自动的变化，来做正确的工作。

◆这里，使用的是 ASCII 而非 EBCDIC。

```
$_ = “The HAL-9000 requires authorization to continue.”;
if(/HAL-[0-9]+/){
  print “The string mentions some model of HAL computer.\n”;
}
```

有时，指出没有被字符类包含的字符更加容易。字符类前使用符号^将取此字符类的补集。也就是说，[^def]]将匹配上这三个字符中之外的任意单个字符。[^n\-z]将匹配上 n, -, z 之外的任何字符。(连接符(-)前面使用反斜线的原因是，它在此字符类中有特别的含义（表示字符的范围：译者注)。但/HAL-[0-9]+/中第一个连接符(-)前不需要反斜线，因为此时的连接符不会被理解为有特殊的含义。)

7．3．1 字符类的简写

有一些字符类出现的非常频繁，因此提供了其简写形式。例如，任何数字的类，[0-9]，可以被简写为：\d。因此，HAL 这

个例子可以被写作/HAL-\d+/。

\w 被称作 "word'字符：[A-Za-z0-9_]。如果你的 "words"由通常的字母，数字，下划线组成，那你将非常喜欢它。通常认为 "word"由字母，连接符(-)，撇号（'）◆组成，我们希望能改变这种定义◆。 因此使用它，请记住我们对 "word"的定义，字母，数字， 下划线组成。

◆至少，在英语中是这样。在其它语言中，其 words 由不同的符号组成。查看 perllocale 的帮助手册了解更多的信息。

◆当查看 ASCII 编码的英语文本时，我们遇到单引号和撇号(')是相同字符的问题，因此很难说 cat'是 cat 和一个撇号(')，还是 cat 后接单引号。这可能是计算机还不能接管世界的一个原因☺。

当然，\w 不能匹配单词，而只能匹配单个字符。为了匹配整个单词，需要后接加号。模式 /fred \w+ barney/将匹配 fred，空格，一个 "单词（word）"，然后是空格和 barney。因此，如果 fred 和 barney 之间有一个单词◆，由单个空格分隔开，它将能匹配上。

◆我们将停止在 word 上加引号；现在你已经知道其是由字母-数字-下划线组成的。

你可能已经注意到在前一例中，如果能更加灵活的匹配空白将很方便。\s 对于匹配空白（whitespace）将非常方便。它等价于[\f\t\n\r]，其含 5 个空白字符：格式符（form-feed）；制表符（tab），换行符，回车，以及空格符。同其它简写符号一样，\s 匹配此类中的单个字符，如果使用\s*将匹配任何个数的空白（包括没有），或者\s+匹配一个以上的空白（事实上，很少见到单独使用\s，而不使用任何的数量词(*, +)）。由于这些空白符看起来类似，因此可以使用这种简写形式，将它们统一处理。

7．3．2 简写形式的补集

某些时候，你可能希望得到这三种简写形式的补集。如果那样的话，你可以使用[^\d], [^\w], 和[^\s]，其含义分别是，非数字的字符，非 word（记住我们对 word 的定义）的字符，和非空白的字符。也可以使用它们对应的大写形式：\D, \W, \S 来完成。它们将匹配它们对应的小写形式不能匹配上的字符。

这些简写形式可以在字符类中使用，或者在大的字符类中的中括号里面使用。也就是说你可以使用/[\dA-Fa-f]+/来匹配十六进制（底为 16）的数字，它将 ABCDEF(或者其小写形式)作为附加的数字（11 到 15）。

另一个类字符[\d\D]，它的意思是任何数字，和任何非数字，则意指任何字符。这是匹配所有字符的一种通用方法，甚至包括换行符，而点(.)匹配除换行符以外的任何字符。而[^\d\D]则完全没用，因为它匹配既非数字也非非数字的字符，那什么也不是。

7．4 练习

参见附录 A 查看下面习题的答案：

记住，正则表达式能完成的工作经常让你吃惊。这也是本章练习比其它章节重要的原因。

1. [10]写一个程序，输出所有提到 fred 的行（不要输出其它行）。如果输入字符串 Fred, fredrick, Alfred，能匹配上吗？准备一个小的文本文件，其中包含如：“fred lintsotne”以及类似的信息。使用这个文本文件作为此程序的输入，以及本节下面练习的输入。

2. [6]修改上面的程序，允许匹配 Fred。现在它能匹配，Fred, fredrick, Alfred 吗?（将这些名字加入输入文件中）

3. [6]写一个程序，输出出现句号(.)的行，忽略其它行。使用前面练习中的文件进行练习：它能找到 Mr. Slate 吗？

4. [8]写一个程序，输出有一个字母大写，而非所有字母都大写的行。它能匹配 Fred，而不匹配 fred 和 FRED 吗？

5. [8]额外的练习：写一个程序，它能输出所有同时提到 wilma 和 fred 的行。

第八章 正则表达式的应用

在前一章，我们讨论了正则表达式。现在你将学习怎样在 Perl 中使用正则表达式。

8．1 使用 m//匹配

我们曾经将模式放在一对正斜线（//）里面，如/fred/。这是 m//（模式匹配）的一种简写。同 qw//操作一样，可以使用任何成对的分隔符。因此，可以使用 m(fred), m<fred>, m{fred}, m[fred]，或者 m,fred,, m!fred!, m^fred^，其它非成对的分隔符也可以◆。

◆非配对分隔符是那些“左”和“右”是相同的，两头使用的是同一个符号。

如果使用正斜线（/）作为分隔符，则可以省略掉前面的 m。由于 Perler 喜欢少输入字符，因此大多数模式使用的是正斜线，如/fred/。

使用一个不会在模式中出现的字符作为分隔符◆。如果想写一个匹配 web URL 开头部分的模式，你可能使用/http:\/\//来匹配 http://。但如果使用 m%http://%将更易于阅读，书写，维护，以及调试。使用花括号（{}）作为分隔符也是很平常的。如果你使用的是专为程序员设计的文本编辑器，由于它能自动从开花括号跳到闭花括号，这对于维护代码将非常有用。

◆如果使用配对的分隔符，那不用当心模式内部会出现这些分隔符，因为通常模式内部的分隔符也是配对的。因此，m(fred(.*)barney), m{\w{2,}}, m[wilma[\n \t]+betty]是正确的。对于尖括号（<和>），它们通常不是配对的。如模式 m{(\d+)\s*>=?\s*(\d+)}, 如果使用尖括号，模式中的尖括号前因当使用反斜线（\），以免模式被过早的结束掉。

◆记住，正斜线不是元字符，如果它不是分隔符，则不需在前面使用反斜线。

8．2 可选的修饰符

有几个修饰符(modifier)，通常叫做标记(flag)，可以后缀在正则表达式后面来该变其默认的行为。

8．2．1 不区分大小写：/i

要创建一个大小写无关的模式，如匹配 FRED 时，也能匹配上 fred, Fred，可以使用修饰符 /i：

```
print “Would you like to play a game?”;
chomp($_ = <STDIN>);
```

```
if(/yes/i) {#大小写无关
    print "In that case, I recommend that you go bowling.\n";
    }
```

8．2．2 匹配任何字符：/s

默认情况下，点(.)不匹配换行符，这对于"单行中查找"的问题能很好解决。如果你的字符串中有换行符，并希望点(.)能匹配它们，那可以使用/s 这个修饰符。它将模式中点(.)◆的行为变成同字符类[\d\D]的行为类似：可以匹配任何字符，包括换行符。从下例中可见其区别：

◆如果你想改变其中的一部分，但不是全部，那可以将此部分用[\d\D]代替

```
$_ = "I saw Barney\ndown at the bowing alley\nwith Fred\nlast night.\n";
if(/Barney.*Fred/s){
    print "That string mentions Fred after Barney!\n";
}
```

如果不使用/s，那么上述模式将不能被匹配上，因为这两个字符不在同一行中。

8．2．3 添加空格：/x

/x 修饰符，允许你在模式中加入任何数量的空白，以方便阅读：

```
/-?\d+\.?\d*/          #这是什么含义？
/ -? \d+ \.? \d* /x     #要好些
```

由于/x 允许模式中使用空白，那么模式中的空格，制表符将被忽略。可以在空格前加上反斜线或者使用\t（许多方法中的一种），但匹配空白更常用的方法是使用\s(\s*或\s+)。

Perl 中，注释可以被作为空白，因此使用/x，可以在模式中加上注释：

```
/
-?     #可选的负号
\d+    #小数点前一个或多个十进制数字
\.?    #可选的小数点
\d*    #小数点后一些可选的十进制数字
/x     #模式结束
```

井号(#)表示后面是注释，如果需要匹配井号，可以使用\#或[#]。注意不要在注释中使用闭分隔符，否则将结束此模式匹配。

8．2．4 将可选字符结合起来

如果在一个模式中需使用不止一个修饰符，可以一个接着一个。其顺序是不重要的：

```
if(/barney.*fred/is/){ # /i 和/s
  print "That string mentions Fred after Barney!\n";
}
```

下面是包含注释的版本：

```
if (m{
  barney     #小个子小伙
  .*         #可以包含任何字符
  fred       #嗓门大的小伙
  }six) {    #修饰符包括包括/s, /i, /x
  print "That string mentions Fred after Barney!\n";
}
```

注意这里的分隔符：花括号，它允许那些专为程序员设计的文本编辑器可以从正表达式的开端跳到末端。

8．2．5 其它选项

还有许多修饰符，将在遇到它们时再讨论。你也可以阅读 perlop 帮助手册中介绍 m//的部分，或者本章后面正则表达式操作的部分。

8．3 锚定

默认情况下，如果模式在字符串开头没能匹配上，它会顺着字符串下去，直到匹配上为止。如果使用了锚定(anchors)则可以要求模式在特定的位置进行匹配。

符号^(脱字字符◆)表示在字符串的开头进行匹配，而符号$则表示在结尾◆。因此，模式/^fred/只匹配字符串的开头部分；它不会匹配上 manfred man。而/rock$/只在结尾处匹配；其不会匹配上 knute rockne。

◆是的，在模式中^，有其它的用途。在字符类中，如果将其放在最前面，将会得到此集合的补集。但在字符类外面，它就成了元字符，从而有了别的含义：表明是字符串的开头。字符就那么几个，有时你不得不重复使用它们

◆事实上，它能匹配字符串的结尾部分，或者换行符处。因此会匹配一个字符串的结尾部分，无论其是否含有换行符。许多人都不担=关心这种区别，但记住/^fred$/能同时匹配上"fred"和"fred\n"。

有时，想同时使用这两个锚定来确保匹配整个字符串。一个经常使用的例子是/^\s*$/，它将匹配一个空行（blank line)。这里的"blank（空白）"，可以是空白符，如制表符，空格等，它们是不可见的。能被匹配上的行看起来都是一样的，因此这

个模式将所有的空白按一种方法来处理。如果没有锚定，则它还会匹配上非空行。

8．3．1 词锚定

锚定不仅仅针对字符串的两头。词界锚定，\b，是针对单词使用的。如/\bfred\b/可以匹配上单词 fred，但不能匹配 frederick, alfred, man fred mann。这同字处理软件中的 “全字匹配(match whole words only)”是类似的。

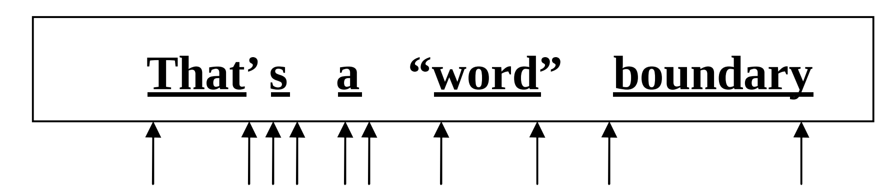
◆某些正则表达式实现中开头的锚定和结尾锚定不同，但 Perl 中均使用\b。

这些单词(words)不是你或者我通常认为的那样；它们是\w 类型，由通常的字母，数字，下划线组成。\b 将从开头或结尾匹配这些\w 类型的字符。

图 8-1 中，每一个“词(word)”下面有灰线，箭头标明\b 匹配的地方。给定字符串总有偶数个词界，因为每一个词均有两个界限(开始和结尾)。

词是字母，数字，下划线组成的序列。这种意义下的词可由/\w+/匹配上。下句中有 5 个词:That,s,a,word,以及 boundary◆。词上的引号不会改变词界的位置；这些词是由\w 字符组成的。

图 8-1　词界：\b



◆这里你可以看到，我们为什么希望能改变 “word”的定义，That’s 应当是一个单词，而不是两个单词，虽然中间有撇号（’）。即便是在通常的英文文本中，上述看法也符合通常的观点，还能找到其它字符的些微差别。

每一个箭头指向下划线的开头和结尾处，因为词界锚定：\b 只匹配这样的词。

词界锚定是非常有用的，如果不希望 cat 匹配上 delicatessen，而 dog 匹配上 boondoggle，或者 fish 匹配上 selfishness。有时，你只想要一个词界锚定，当使用 /\bhunt/ 将匹配上像 hunt, hunting, hunter，这样的单词，但不会匹配 shunt，而/stone\b/将匹配 sandstone, flintstone， 而不能匹配上 capstones。

非词界锚定为\B。它将在任何非\b 匹配的点上进行匹配。因此，模式/\bsearch\B/将匹配 searches, searching, searched, 但不能匹配 search，或者 researching。

8．4 绑定操作符，=~

对$_进行匹配只是默认的行为，使用绑定操作符(=~)将告诉 Perl 将右边的模式在左边的字符串上进行匹配，而非对$_匹配。例如：

```
my $some_other = "I dream of betty rubble.";
    if($some_other =~ /\brub/){
        print "Aye, there's the rub.\n";
}
```

第一次见到绑定操作符(binding operator：=~)，可能觉得它有些像赋值操作符，但它不是。其含义是："这个模式默认将对$_进行匹配，但此时将对左边的字符串进行匹配"。如果没有绑定操作符，则此表达式将对$_匹配。

在下面的例子中（有些不寻常），$likes_perl 将根据用户的输入而得到一个 boolean 值。它有一些 quick-and-ditry，因为输入行很快就被丢弃了。这段代码将读入一行，由右边的模式进行匹配，然后丢弃此输入◆。它没有使用$_。

◆输入的字符不会自动存储在$_中，除非行输入操作（<STDIN>）单独出现在 while 循环的条件判断部分。

```
print "Do you like Perl? ";
my $likes_perl = (<STDIN> =~ /\byes\b/i);
…       #Times passes…
if($likes_perl){
   print "You said earlier that you like Perl, So…\n";
…
}
```

由于绑定操作有非常高的优先级，因此，模式测试部分的括号不是必需的，下面的代码和上面代码的含义是一样的。它将测试部分的结果（而非行输入）返回给变量$likes_perl：

```
my $likes_perl = <STDIN> =~ /\byes\b/i;
```

8．5 模式内的内插

正则表达式可以被内插，如同双引号字符串一样。。这允许我们快速地写出类似 grep 的程序：

```
#! /usr/bin/perl –w
my $what = "larry";

while(<>){
  if(/^($what)/){           #在字符串前面进行匹配
    print "We saw $what in beginning of $_;";
    }
```

当程序运行时，模式将被$what 的值代替。在本例中，它同使用/^(larry)/是一样的，在行的开头处查找 larry。

如果没有在程序中具体指出$what 的值，也可以在命令行中输入，然后使用参数@ARGV：

```
my $what = shift @ARGV;
```

如果命令行中第一个参数是 fred|barney，则模式为 /^(fred|barney)/：在行首查找 fred 或者 barney◆。括号（在查找 larry 的例子中不是必须的）是非常重要的，如果没有它，则在行首查找 fred，或在此行中任意位置查找 barney。

◆机敏的读者知道在命令中通常不能输入 fred|barney, 因为竖线是 shell 的元字符(shell metacharacter)。查看你的 shell 文档，了解怎样将命令行中的参数括起来。

使用上面的代码，可以通过@ARGV 来得到模式，现在的程序有点 Unix 的 grep 命令的味道了。但我们要小心元字符。如果$what 的值为 'fred(barney'，这个模式是/^(fred(barney)/。上述模式是不能工作的，会由于错误使用正则表达式而使程序垮掉。利用一些高级技术◆，你可以捕捉到这种错误（或者在开始时就阻止这些元字符），而不至于让程序垮掉。但现在，只需知道如果用户被赋予正则表达式的强大能力，则他们有责任正确的使用它们。

◆ 可以使用 eval 块来捕捉错误，或者使用 quotemea(或者其等价形式：\Q)将内插部分引用起来，使之不会被当作正则表达式来处理。

8．6 匹配变量

我们曾经在模式中使用过括号，使用括号是由于它可以将模式的某一部分组合起来。同时括号也会引起正则表达式分配新的内存块。这些内存含有括号中的模式所匹配的字符串。如果有不止一对括号，那就不止一块内存块。每一个内存块内有一段字符串，而非模式的一部分。

由于这些变量含有字符串，那它们是标量变量；在 Perl 中,它们具有像 $1, $2 这样的名字。变量个数同模式中括号对数的个数是相同的。如$4 是指第四对括号所匹配的字符串◆。

◆这和后引用(backreference)\4 在模式匹配中引用字符的字符串相同。但它们不仅是同一事物的两个不同名字；\4 是模式正在匹配是引用的；而$4 是模式匹配完成后再引用的。想了解更多的关于 backreferences 的信息，可参见 perlre 的帮助手册。

这些匹配变量（match variables）是组成正则表达式强大功能的重要部分，它允许取出相应的字符串：

```
$_ = “Hello there, neighbor”;
  if(/\s(\w+),/){                    #空格和逗号之间的词
  print “the word was $1\n”;         #the word was there
  }
```

也可以一次使用多个：

```
$_ = “Hello there, neighbor”;
  if(/(\S+) (\S+), (\S+)/){
  print “words were $1 $2 $3”;
}
```

其输出为 words were Hello there neighbor。注意输出中没有逗号。因为第二块内存中没有逗号。使用这种技术，可以选择我们感兴趣的部分。

匹配变量可能是空的◆，如果其没有被匹配上。也就是说，匹配变量的值可能为空串：

◆这和 undefined 是不同的。如果模式中只有 3 个或者更少的括号，那$4 为 undef。

```
my $dino = "I fear that I'll be extinct after 1000 years.";
if ($dino =~ /(\d*) years/) {
  print "That said '$1' years.\n";    # 1000
}

my $dino = "I fear that I'll be extinct after a few millions years.";
if ($dino =~ /(\d*) years/) {
    print "That said '$1' years.\n";    #  空串
}
```

8．6．1 内存值的保存

这些匹配变量的值会保持不变，直到下一个模式成功匹配为止◆。也就是说，一个没有匹配成功的模式将不会改变内存中相应的值，但一个匹配上的模式将重写此内存。这明确的告诉你，不要随意的使用这些变量，除非明确知道它们匹配正确；否则，你可能得到上个模式匹配的结果。下面的例子（不好的例子）本意是输出被$wilma 变量匹配的字符串。但如果匹配失败，它将输出$1 中以前所匹配上的字符串。

◆这里的作用域规则相当复杂（可查看相关文档），除非你希望这些匹配变量在数行后还被使用，否则不会有什么问题。

```
$wilma =~ /(\w+)/;                  #不好，没有检测匹配的结果
print “Wilma’s word was $1… or was it?\n”;
```

这也是为什么模式匹配几乎都在 if 和 while 循环的条件判断出现的原因：

```
if ($wilma =~ /(\w+)/){
     print “Wilma’s word was $1.\n”;
     } else {
     print “Wilma doesn’t have a word.\n”;
  }
```

由于内存中的值不会一直保留，那应当在模式匹配后尽快地(几行内)使用像$1 这样的变量。如果维护人员在正则表达式和使用$1 的表达式之间加入了新的正则表达式，那此时$1 的值为第二个匹配的结果，而非第一个。由于这个理由，如果要在后面使用这个变量的值，应当将其拷贝到普通变量之中。这样做同时也可以使程序更易于阅读：

```
if($wilma =~ /(\w+)/){
      my $wilma_word = $1;
      …
  }
```

在第九章，你将学习到如何在模式匹配时直接将内存中的值存入变量之中，而不需要明确的使用$1。

8．6．2 自动匹配变量

还有三个匹配变量，你可以不花什么代价的使用它们◆，无论这些模式是否有括号。这是个好消息；但同时，这些变量的名字相当古怪。

◆是的。没有完全免费的东西。它们 “免费”只是说不需要匹配的括号而已。我们将在后面讲述其花费的代价。

如果能将这些变量名取得简单一些，如$gazoo，$ozmodiar，Larry 肯定会非常乐意。但在你自己的程序中你可能使用这些变量名。为了让普通的 Perl 程序员在记住 Perl 所有的特殊变量之前能写程序◆，Larry 将大量 Perl 内嵌的变量名取得相当古怪，并且不符合变量名的命名规则。这里提到的三个变量名为：$&, $`, $'。它们看起来相当奇怪，但无论怎样，这就是其名字◆。匹配上的那部分字符串将自动存储在$&之中。

◆同时也应当避免使用像$ARGV 这样的变量，但这些极少数的变量全是大写的。Perl 所有内嵌变量均可在 perlvar 的帮助手册中找到。

◆如果不喜欢这些变量名，你可以使用 English 这个模块，它试图将所有 Perl 奇怪的变量名给与一个普通名字。但使用这个模块的人很少。相反的，Perl 程序员开始越来越喜欢这这些有标点符号的变量名了，管它样子是否古怪呢。

```
if(“Hello there, neigbor” =~ /\S(\w+),/){
  print “That actually matched ‘$&’.\n”;
  }
```

匹配的部分是 “ there,”（空格，单词， 和一个逗号）。变量$1 中的值为 there，而$&为整个被匹配的部分。

匹配部分的前一部分存放在$`之中，后一部分被存到$'。另一种说法是，$`中含有正则表达式引擎在匹配成功前所找到的变量， 而$'为此模式还没有匹配的剩余部分。如果将这三个变量放在一起，你将得到原始字符串：

```
if (“Hello there, neighbor” =~ /\S(\w+),/){
  pirnt “That was ($`)($&)($’)”;
}
```

输出的消息为(Hello)(there,)(neighbor)，为这三个自动匹配变量的值。三个变量的值可能是空的，和之前数字匹配变量的例子一样。它们和数字匹配变量有相同的作用域。通常， 在下次成功匹配前其值不变。

现在，我们讨论我们之前说的“免费”问题。是的，自由是要代价的。这里的代价是，如果你使用了这三个自动匹配变量中的任意一个，无论在程序的什么地方，其它地方的正则表达式的运行数度会变慢一些。虽然，变慢的程度不大，但已经足够让人担忧，因此许多 Perl 程序员从不使用这些自动匹配变量◆。相反的，使用的替代的方法。例如，如果需要使用$&，那么在整个模式上加上括号，并使用$1 代替。

◆许多人没有测试其程序的实际效率，以判断其替代方法是否节约了时间。这些变量看起来是有害的， 但我们不因该在实际比较过它们的效率之前而责备它们；许多程序从这个三个变量中获益，这些程序一周只运行几分钟，优化和评估其效率是浪费的。但，如果我们对一毫秒都非常关心呢？顺便提一句，Perl 开发者正着手解决这一个问题，但在 Perl6 之前可能都不会有好的答案。

匹配变量（自动的和编号的）经常在需要替换的地方使用，这在下一章中介绍。

8．7 一般的数量词

模式中的数量词表示前面项重复的次数。你已经见过三个数量词：*, +, ?。如果这三个数量词仍不能满足你的要求，那可以使用花括号({})，花括号中有 2 个数字，由逗号隔开，表示前面一项允许重复的次数。

模式/a{5,15}/将匹配 5 个到 15 个 a 中的任意一个（包括 5，和 15）。如果 a 出现了 3 次，则次数太少，而不能匹配上；如果出现 5 次，则匹配上了；如果出现 10 次，仍然匹配上。如果出现 20 次，仍将匹配上，前 15 个将匹配上。

如果省略掉第二个数字（逗号留下），现在没有上限了。因此，/(fred){3,}/将在一行中有 3 个或者更多个 fred 时匹配上（fred 之间不允许有其它字符，包括空格）。这里没有上限，因此如果有 88 个 fred，仍将匹配上。

如果除了上限数字外，逗号也被省略了，那将匹配确定的次数：/\w{8}/将严格的匹配 8 个 word（字母，数字，下划线）（可能被其中一个长字符串部分所匹配上）。/,{5}chameleon/将匹配“,,,,,chmeleon”。

前面介绍过的三个数量词是简写形式。星号(*)等同于{0,}，表示 0 个或多个。加号(+)等同于{1,}，表示 1 个或多个。而问号(?)则等同于{0,1}。在实际程序中，很少使用花括号的数量词，前面介绍的三个数量词(*,+,?)基本已能应付。

8．8 优先级

正则表达式中的这些元字符，可能让你觉得如果没有一个表很难记住它们的关系。这就需要一个优先级的表，表明模式中哪一部分应当结合在一起。这和操作符的优先级表有些不同，正则表达式的优先级表更简单，只有 4 个级别。本节，将回顾 Perl 在模式中使用的所有的元字符。

1．在此优先级表的最顶端是括号：（( )），在分组和引用内存值的时候使用。括号内部的任何部分比括号外的部分结合更紧密。

2．第二级是数量词。这里有星号(*)，加号（+），问号（？）以及由花括号表示的数量词，如{5,15}, {3, }, {5}等。它们通常和前一项元素结合。

3．第三级的是锚定和序列（sequence）。锚定包括(^) 表明字符串的开头，($)表明结尾，(\b)词界符，(\B)非词界符。序列（一个元素紧接着一个元素）实际上是一种操作，虽然它没有使用元字符。这段话的含义是一个单词中的字母结合更紧密，就像锚定紧贴字母一样。
4．优先级最低的是竖线(|)，表示或。由于其优先级最低，它通常将模式划分成几个部分。它在优先级最底端是因为我们希望像|/fred|barney/里面的字母比或(|)结合更紧密。如果或(|)的优先级比序列的优先级更高,那么，上述模式的含义是匹配 fre，接着是 d 或者 b，然后是 arney。因此，或(|)的优先级最低，字母序列的优先级要高些。

除了优先级表外，还有被称为原子（atoms）的东西，它们组成模式最基本的块。它们是单个字符，字符类，以及后引用(backreference)。

8．8．1 优先级练习

当需要解析复杂的正则表达式时，你应当像 Perl 那样使用优先级表来理解它。

例如，/^fred|barney$/很可能并不是程序员想要的模式。由于竖线(|)的优先级最低；它将上述模式分成了两部分。这个模式在或者开头是 fred,或者结尾是 barney 时匹配上。很可能程序员想要的是(^(fred|barney)$),这将匹配只有 fred 或者只有 barney 的行◆。那/(wilma|pebbles?)/呢？问号也对前面的元素有效◆，它将匹配 wilma, pebbles, 或者 pebble，可能是某个长字符串的一部分（因为没有锚定）。

◆当然，结尾处可以有换行符，我们在早期讨论$锚定时提到过。

◆因为数量词?和字母 s 结合更紧密

模式/^(\w+)\s+(\w+)$/将匹配那些含有一个“词（word）”，一些空白，另一个“词(word)”，之外没有其它符号的行。它将匹配上像 fred flintstone 这样的行。模式中的括号是不需要的，当然它们可以将匹配上的字符串存放在$1, $2 之中。

当解析复杂的正则表达式时，括号将有助于你区分优先级。可以这样做，但请记住，如果使用了括号，则这些括号所匹配的字符串将放在相应的内存变量之中($1, $2. . .)之中。当添加新的括号时也可能需要改变这些以前的编号◆。

◆察看 perlre 的帮助手册了解更多的不会引起内存分配的括号，它们仅是用来分组。

8．8．2 更多

虽然已经涵盖了程序员日常编程所需要知道的正则表达式特性的绝大部分，但还没介绍完。还有一些在羊驼书中有介绍。也可以察看 perlre, perlrequick,以及 perlretut 的帮助手册来了解更多信息，以及 Perl 模式所能完成的工作◆。

◆查看 CPAN 中的 YAPE::Regexp::Explain。

8．9 模式测试程序

现在，程序员已经能够书写正则表达式了，虽然不容易搞清此正则表达式所能完成的工作。通常发现正则表达式实际匹配的比要求的更多，或者更少；或者比预期的更早，更晚，或者完全不匹配。

下面的程序用来测试一个模式，看它能匹配什么以及在什么地方匹配：

```
#! /usr/bin/perl
 while(<>){                                      #一次取一行输入
   chomp;
   if(/YOUR_PARTTEN_GOES_HERE/){
```

```
    print "Matched: |$`<$&>$'|\n";    #特殊的变量
}else{
    print "no match: |$_|\n";
}
```

这个模式测试程序是给程序员用的，而非终端用户，从它没有提示信息这点可以看出。它会将输入行和模式 YOUR_PARTTERN_GOES_HERE 进行比较。对于任何匹配上的行，它使用三个特殊变量($`, $&, $')来指出是在什么地方匹配上的。如果模式为/match/，输入为/beforematchafter/，你将看到如下信息："Matched: |before<match>after|",尖括号标明匹配上的部分。你能立刻看到匹配上的行是否是你所希望的。

8．10 练习

参考答案在附录 A 中。

其中有几个例子要求你使用本章的测试程序。你可以用它，但请注意这些符号◆，不要输错了。

◆如果手工输入它，记住反引号(backtick) 字符(`)同撇号(apostrophe)（'）是不一样的。在当今大多数键盘上（至少，U.S 键盘上）反引号正好在数字键 1 的左边

1. [8]使用模式测试程序。创造一个模式能匹配字符串 match。使用字符串 beforematchafter 进行测试。输出结果将其三部分放在正确位置了吗？

2. [7]使用模式测试程序，创造一个模式能匹配任何单词（\w 意义下的单词），但这个单词必需以字母 a 结尾。它匹配 wilma 而没匹配 barney 吗？它匹配 Mrs. Wilma Flintstone 吗? wilma&fred 呢？使用前一章习题的文件进行练习（如果没有上述字符串，则加上它们）

3. [5]修改第二题的程序，使之将由 a 结尾的单词放到$1 之中。同时修改源代码，使此变量对应的值被放在单引号之中，如$1 contains 'Wilma'。

4. [5]额外练习：修改第三题程序，使之能捕捉由 a 结尾的单词之后的 5 个字符（如果有那么多），并将之放入一个独立变量中。例如，如果输入的是 I saw Wilma yesterday，则紧接的 5 个字符是 yest（前有空格）。如果输入是 I, Wilma!，则只有一个字符。它现在还能匹配 wilma 吗？

5. [5]写一个程序（不是测试程序），能输出任何由空白结尾的输入行（非换行符）。在输出的结尾处放置一个标记符，使之能标记出空白。

第九章 使用正则表达式处理文件

可以通过正则表达式来改变文本。到目前为止，只介绍过怎么匹配模式。现在，我们将向你演示怎样通过模式来改变字符串中相应地部分。

9．1 使用 s///进行替换

如果将 m// 这个模式匹配看作同文字处理器的“查询（search）”类似的功能，那 Perl 中 s ///操作的则类似于“查询并替换（search and replace）”。它将替换变量中◆模式所匹配上的部分：

◆同 m//不一样，m//可以和任何字符串表达式进行匹配，s///只能修改被称为左值（lvalue）的数据。这通常是一个变量，虽然它可以是任何可在赋值符左侧出现的东西。

```
$_ = “He’s out bowling with Barney tonight.”;
s/Barney/Fred/;                          #Barney 被 Fred 替换掉
print “$_\n”;
```

如果没有匹配上，则什么也不会发生，此变量也不会有任何更改：

```
#接上例：现在 $_ 为“He’s out bowling with Fred tonight.”
s/Wilma/Betty/;          #用 Wilma 替换 Betty（失败）
```

模式和被替换的字符串可能更复杂。下例中，替换的字符串使用了变量：$1，其值由模式匹配所赋值：

```
s/with (\w+)/agaist $1’s team/;
print “$_\n”;            #为 “He’s out bowling against Fred’s team tonight”;
```

下面还有些其它可能的替换方法。(它们在这里只是作为例子出现。在实际代码中，很少在一行中做这么多不相关的替换。)

```
$_ = “green scaly dinosaur”;
s/(\w+) (\w+)/$2, $1/;       #现在为 “scaly, green dinosaur”;
s/^/huge, /;                 #现在为 “huge, scaly, green dinosaur”

s/,.*een//;                  #空替换,现在为 “huge dinosaur”
s/green/red/;                #匹配失败,仍然为 “huge dinosaur”
s/\w+$/($`!)$& /;            #现在为 “huge (huge !)dinosaur”
s/\s+(!\W+)/$1 /;            #现在为 “huge (huge!) dinosaur”
s/huge/gigantic/;            #现在为 “gigantic (huge!) dinosaur”
```

s///会返回一个 Boolean 值。如果成功替换则返回 true；否则返回 false：

```
$_ = “fred flintstone”;
if(s/fred/wilma/){
  print “Successfully replaced fred with wilma!\n”;
}
```

9．1．1 使用/g 进行全局替换

在前面的例子中你可能已经注意到，s///值进行一次替换，无论是否还有地方还能匹配上。当然，这只是默认的行为。修饰符/g 要求 s///将不相重叠◆的所有匹配上的部分都进行替换：

◆它是不相重叠的，因为每一次新的匹配都是从最近匹配成功的地方之后开始进行的。

```
$_ = “home, sweet home!”;
s/home/cave/g;
print “$_\n”;                    # “cave, sweet cave!”;
```

全局替换的一个常用地方是将多个空格用单个空格替换掉：

```
$_ = “Input     data\t may have      extra whitespace.”;
  s/\s+/ /g;                                      #现在是 “Input data may have extra whitespace.”
```

现在已经知道怎样去掉多余的空格，那怎样去掉开头和结尾的空白呢？这是非常容易的：

```
s/^\s+//;                          #将开头的空白去掉
s/\s+$//;                          #将结尾的空白去掉
```

我们可以使用/g，只用一步来完成，，但这可能影响效率，至少在我们写作此书时是这样。正则表达式通常是可被优化的，想了解更多的信息，可以参看《掌握正则表达式》(Mastering Regular Expressions)（O’Reilly），其中有关于如何使正则表达式更快(或者更慢)的讨论。

```
s/^\s+|\s+$//g;        #将开头，结尾的空白去掉
```

9．1．2 不同的分隔符

如同 m//和 qw//一样，我们也可以改变 s///的分隔符。但这里使用了 3 个分隔符，因此有些不同。

通常的（非配对的）字符，由于没有左字符和右字符的区别，则可以像使用正斜线(/)那样使用。如，使用井号（#）◆作为分隔符：

◆这里，我们需要向我们的英国的朋友道歉，因为井号(#)(在英语中#号的单词时 pound,有磅的含义，译者注)对于他们有别的含义。虽然井号在 Perl 中也表示注释的开始，但在替换(s)之后，由于 Perl 期望一个分界符，因此不会将#号当作注释的提示符处理。

```
s#^https://#http://#;
```

如果使用的是配对的字符，也就是说其左字符和右字符不的，则必需使用两对：一对存放模式，一对存放替换的字符串。此时，分隔符甚至可以是不同的。事实上，分隔符还可以使用普通的符号（非配对的）。下面三例是等价的：

```
s{fred}{barney};
s[fred](barney);
s<fred>#barney#;
```

9．1．3 可选的修饰符

除了/g 修饰符外◆，替换操作中还可以使用/i, /x, 和 /s，这些在普通的模式匹配中已经出现过的修饰符。其顺序是无关紧要的。

◆我们这里所说的是像"/i"这样的修饰符，但分隔符有可能不是/。

```
s#wilma#Wilma#gi;            #将所有的 WilmA,或者 WILMA 等等，由 Wilma 替换掉
s{__END__.*}{ }s;           #将 END 标记及其后面的行去掉
```

9．1．4 绑定操作

同 m//一样，我们也可以通过使用绑定操作符改变 s///的替换目标：

```
$file_name =~ s#^.*###s;                  #将$file_name 中所有的 Unix 类型的路径去掉
```

9．1．5 大小写转换

有时，希望确保被替换的字符串均是大写的（或者不是，视情况而定）。这在 Perl 中只需使用某些修饰符就能办到。\U 要求紧接着的均是大写：

```
$_ = "I saw Barney with Fred.";
s/(fred|barney)/\U$1/gi;                     #$_现在是 "I saw BARNEY with FRED."
```

同样，也可以要求后面的均为小写：\L：

```
s/(fred)|barney/\L$1/gi;                     #$_现在是 "I saw barney with fred."
```

默认时，会影响到剩余的（替换的）字符串。可以使用\E 来改变这种影响：

```
s/(\w+) with (\w+)/\U$2\E with $1/I;        #$1 现在是 “I saw FRED with barney.”
```

使用小写形式时(\l 和\u)，只作用于下一个字符：

```
s/ (fred|barney)/\u$1/ig;        #$_现在是 “I saw FRED with Barney.”
```

也可以同时使用它们。如使用\u 和\L 表示“第一个字母大写，其它字母均小写”◆：

◆\L 和\u 可以按任意顺序出现。Larry 意识到人们有时可能按相反顺序使用它们，因此他将 Perl 设计成，在这两种情况下都是将第一个字母大写，其余的小写。Larry 是个非常好的人。

```
s/(fred|barney)/\u\L$1/ig;        #$_现在为 “I saw Fred with Barney.”
```

这些在替换中出现的大小写转换的修饰符，也可在双引号中使用：

```
print “Hello, \L\u$name\E, would you like to play a game?\n”;
```

9．2split 操作

另一个使用正则表达式的操作是 split，它根据某个模式将字符串分割开。这对于由制表符分割开，冒号分割开，空白分割开，或者任意字符分割开的数据是非常有用的◆。任何可在正则表达式之中（通常，是一个简单的正则表达式）指定分离符（separator）的地方，均可用 split。其形式如下：

◆不包括“comma-separated values（逗号分割开的数据）”，其一般被称作 CSV 文件。使用 split 对它处理是非常痛苦的；最好利用 CPAN 中的 Text:CSV 模块

```
@fields = split /separtor/, $string;
```

split◆将模式同字符串进行比较，将由分离符所分隔开的子串作为列表返回回来。当模式匹配上，既是此次匹配结束，和新匹配的开始。因此，任何匹配模式的部分均不会在返回值中出现。下面是一个典型的 split 操作，由冒号分开：

◆它是一个操作，虽然其行为很像函数，通常每一个人都认为它是一个函数。但其技术上的区别超出了本书讨论的范围。

```
@fields = split /:/, “abc:def:g:h”;        #返回(“abc”, “def”, “g”, “h”)
```

可能得到空的元素，如果其中有两个分隔符是连在一起的：

```
@fields = split /:/, “abc:def::g:h”;        #得到(“abc”, “def”, “”, “g”, “h”)
```

这里有一条规则：开头的空元素会被返回，但结尾的空元素被丢弃◆。虽然第一次见到时看起来有些古怪，但很少引起问题。

◆这仅是默认的行为，是基于效率方面的考虑。如果担忧丢失了结尾的空元素，可以使用 split 的第三个参数: -1。请参阅 perlfunc 的帮助手册。

```
@fields = split /:/, “:::a:b:c:::”;      #得到（“”, “”, “”, “a”, “b”, “c”）;
```

使用空白/\s+/这个模式进行分割是非常常见的。在这个模式下，所有的空白等价于单个空格：

```
my $some_input = “This    is a \t            test.\n”;
my @args = split /\s+/, $some_input;                     #(“This”, “is”, “a”, “test.”)
```

默认时，split 对$_操作，模式为空白：

```
my @fields = split;       #同 split /\s+/, $_;
```

这同使用模式/\s+/基本上是一样的，除了，开头的空元素被丢弃。因此，如果某行由空白开头，则它们将不会出现在返回的列表中之中（如果想得到同以空白为分离符的 split 操作相同的结果，可在模式的部分使用单个的空格：split ‘ ’, $other_string。使用空格而非模式，是另一种特殊类型的 split 操作）。

通常，split 所使用的模式同这里出现的一样简单。但如果模式变得复杂时，那应避免在模式中使用括号。请参阅 perlfunc 以了解更多的信息◆。

◆你可能需要察看 perlre 的帮助手册中关于什么情况下括号只是起到分组作用的信息。

9．3join 函数

join 函数不使用模式，但它完成同 split 相反的操作：split 将一个字符串分割开，而 join 函数将这些分割的部分组合成一个整体。join 函数类似于：

```
my $result = join $glue, @pieces;
```

join 函数的第一个参数是粘合元素(glue)，它可以是任意字符串。剩下的参数是要被粘合的部分。join 将粘合元素添加在这些部分之间，并返回其结果：

```
my $x = join “:”, 4, 6, 8, 10, 12;    #$x 为 “4:6:8:10:12”
```

在本例中，我们有五个元素，因此有 4 个冒号或者说粘合元素。这些粘合元素只在这些粘合部分之间出现，而不会在之前或之后出现。因此粘合的元素要比粘合的部分的个数少 1。

这意味着如果列表中元素个数小于 2，则不会有粘合的元素：

```
my $y = join “foo”, “bar”;                   #得到 “bar”
my @empty;                                   #空数组
my $empty = join “baz”, @empty;              #没有元素，因此为空串
```

使用上面的$x，我们可以先将一个字符串分割开，再使用不同的分隔符（粘合元素）将它们重组起来：

```
my @values = split /:/, $x;     #@values 为(4, 6, 8, 10, 12)
my $z =join “-”, @values;     #$z 为 “4-6-8-10-12”
```

split 和 join 可以一起使用，但不要忘了 join 的第一个参数是字符串，而非模式。

9．4 列表上下文中的 m//

当使用 split 时，模式指定了分离符：这一部分不是有用的数据。有时指定要保留的部分更容易。

在列表 context 中使用模式匹配(m//)时，如果匹配成功返回值为内存变量值的列表；如果匹配失败则为空列表：

```
$_ = “Hello there, neighbor!”;
my($first, $second, $third) =/(\S+) (\S+), (\S+)/;
print “$second is my $third\n”;
```

这种方法使我们可以给这些匹配的变量以合适的名字，这些值不会由于下次模式匹配而被覆盖（由于代码中没有=~,模式会自动（默认行为）和$_进行匹配）

在 s///中介绍的/g 修饰符也可在 m//中使用，它允许你在字符串中的多处进行匹配。在这里，由括号括起来的模式将在每一次匹配成功时返回其内存中所存放的值：

```
my $text = “Fred dropped a 5 ton granite block on Mr. Slate”;
my @words = ($text =~ /([a-z]+)/ig);
print “Result: @words\n”;
#Result: Fred dropped a ton granite block on Mr slate
```

这同使用 split 有些类似。这里不是指定我们要去掉的部分，而是指定我们要保留的部分。

如果有不止一对括号，每一次返回不止一个字符串。例如将字符串放入 hash 中，如下：

```
my $data = “Barney Rubble Fred Flintstone Wilma Flintstone”;
my %last_name = ($data =~ / (\w+)\S+(\w+)/g);
```

每当模式匹配成功时，将返回一对值。这些一对一对的值就成了 hash 中的 key/value 对。

9．5 更强大的正则表达式

通过这三章对正则表达式的学习，你已经知道了它的强大功能。正则表达式在 Perl 中处于核心地位。Perl 开发者还添加了

更多的功能，本节中你将了解其中最重要的部分。同时你将学习到一些关于正则表达式引擎的内部操作。

9．5．1 非贪婪的数量词

已经出现过的四个数量词（在第七章和第八章）均是贪婪的。这就是说它们将最大限度的匹配，而非在匹配成功时即立刻返回。下面是一个例子：假设在 fred and barney went bowling last night 上使用/fred.+barney/进行匹配。我们知道正则表达式将匹配上，下面我们具体的讲解这一个过程◆。首先，子模式 fred 将匹配其对应的字符串。模式的下一部分是 .+，它将匹配除了换行符之外的任意字符，次数大于等于一。。但，由于加号(+)是贪婪的；它将尽可能的进行匹配。因此，它将匹配剩余的所有字符串，包括 night。这可能让你惊奇，但故事还没结束。）

◆正则表达式引擎作了一些优化，真实情况和我们这里讲述的有些不同，这些优化随 Perl 的不同版本而可能不同。但也不能说其操作过程和我们这里讲述的不同。想了解其具体的操作过程，请阅读最新的源码。注意给你发现的 bug 提交补丁。

现在对 banrey 进行匹配，但不能成功，因为已经到了字符串的结尾处。由于 .+在少一个字符的情况下仍能匹配成功，因此它退回字符串最后一个字母 t。（它虽是贪婪的，但更希望整个模式能匹配成功。）

子模式 barney 又尝试匹配，结果仍是不行。因此 .+再退回字母 h，又进行匹配。一个字符接一个字符，.+退回其匹配的字符，直到其退回了字符串 barney。最后，子模式 banrey 被匹配上了，现在整个模式都匹配上了。

正则表达式引擎有大量的像上面那样的回退（backtracking）操作，尝试每一种可能，直到成功或者根本不能匹配为止◆。如本例所显示的那样，这些操作引起了大量的回退操作，因为这些数量词匹配了太多的字符串，正则表达式引擎强迫它们返回一些。

◆有些正则表达式引擎所采用的方法不同，一旦匹配上则进行下一项的匹配。但 Perl 的正则表达式引擎更关心模式是否匹配，因此找到所有能匹配模式的字符串，引擎工作才算完成。请参看 Jeffy Friedl's 的书《掌握正则表达式》(Mastering Regular Expressions)(O'Reilly)。

因此对于每一个贪婪的数量词，需要一种非贪婪的方法。不是使用加号(+),而是使用非贪婪的数量词 +?，它将匹配一次或多次（加号的意思），但其匹配尽可能少的次数，而非尽可能多的次数。现在我们来看看模式为/fred.+?barney/时的过程。

首先，fred 将被匹配上。接着，模式的下一部分是 .+?，它匹配的字符个数不大于 1，因此匹配 fred 后面的空格。下一个子模式是 banrey，它在这里不能被匹配（因为现在的位置是 and barney…的开头)。.+?再匹配 a，剩下的模式继续进行匹配。又一次，barney 不能匹配上，因此 .+?再匹配 n，依次类推。当 .+?匹配了这 5 个字符后，barney 可以被匹配上了，现在模式匹配成功。

这里也存在一些回退操作，但由于引擎只需回退，并只尝试几次，其在速度上会有很大提高。但，这种提高依赖于 banrey 在 fred 的附近能被找到。如果数据中 fred 在字符串的开头，而 barney 在结尾处，则贪婪数量词方法的速度更快。因此，正则表达式的速度依赖于具体的数据。

非贪婪数量词不仅和效率相关。即便它和其对应的贪婪数量词表达式均能匹配（或者不能匹配）同一个字符串，它们匹配的部分也可能是不同的。例如，假设你有一些 HTML 类型的◆文本，你想移除标记<BOLD>和</BOLD>，而保留其间的内容。下面是文本：

◆我们不能使用真实的 HTML，因为通过简单的正则表达式不能对其进行解析。如果想对 HTML 或者某种简单的标记语言进行解析，使用模块来处理更加恰当。

```
I'am talking about the cartoon with Fred and <BOLD>Wilma</BOLD>!
```

下面是一种移除标记的方法。它有什么错误呢？

```
s#<BOLD>(.*)</BOLD>#$1#g;
```

其问题出在星号是贪婪的◆。如果文本变成了下面的样子，会得到什么结果?

◆还有另一个问题：我们不得不使用/s 修饰符，因为结束的标记可能和开始的标记不在同一行。幸好我们这里只是练习；在实际代码中，最好使用模块。

```
I thought you said Fred and <BOLD>Velma</BOLD>, not <BOLD>Wilma</BOLD>
```

此时，模式将匹配从第一个<BOLD>到最后一个</BOLD>之间的内容，中间的部分被保留下来。噢！我们需要非贪婪的数量词。星号的非贪婪的类型是*?，因此此模式应当是：

```
$#<BOLD>(.*?)</BOLD>#$1#g;
```

现在它能正确执行了。

由于加号的非贪婪类型是 +?，星号的为 *?，你可能已经意识到剩下的两种数量词其对应的类型也是类似的。花括号的非贪婪类型看起来一样，只是在闭花括号后有一个问号，如{5,10}?或者{8,}?◆。甚至问号数量词也有非贪婪类型：??。它匹配一次或者 0 次，但倾向于匹配 0 次。

◆理论上，对于单个的数字，也应有非贪婪类型，如{3}?。由于它是指匹配前面一项 3 次，则没有任何意义说其是贪婪或者非贪婪的。

9．5．2 匹配多行文本

通常，正则表达式是针对单行文本的。由于 Perl 可以处理任意长度的字符串，因此，Perl 的模式可以轻易的对多行文本进行匹配，就像单行文本一样。当然，表达式中应当包含多行文本。下面的字符串中有 4 行：

```
$_ = “I’am much better\nthan Barney is\nat bowling,\nWilma,\n”;
```

锚定^和$是指整个字符串的开头和结束（参阅第八章）。但/m 这个正则表达式选项允许它们根据内部的换行符进行匹配（将 m 看作多行（think m for multiple lines））。这时锚定针对每一行，而非整个字符串。因此，这个模式可以匹配上：

```
print “Found ‘wilma’ at start of line\n” if /^wilma\b/im;
```

同样的，在多行字符串中，也可以分别针对单行进行替换。在下面的例子中，我们将整个文件读入一个变量之中◆，然后

将文件名字加在每一行的开头处：

◆希望它很小。我们指的是文件，而非变量名。

```
open FILE, $filename
   or die "Can't open '$filename': $!";
my $lines = join '', <FILE>;
$lines =~ s/^/$filename: /gm;
```

9．5．3 更新大量文件

更新文件最常用的方法是写一个和以前的文件相似的新文件，我们可以根据的需要进行修改。如你所知，这和对同一个文件上进行更新的结果类似，但上述方法有一些副作用。

本例中，我们有相似格式的上百个文件。其中一个是 fred03.dat，如下：

```
Promram name: granite
Author: Gilbert Bates
Company: RockSoft
Department: R&D
Phone: +1 503 555-0095
Date: Tues March 9, 2004

Version: 2.1
Size: 21k
Status: Final beta
```

我们希望修改这个文件，使之含有不同的信息。下面是我们希望修改后它所呈现的样式：

```
Program name: granite
Author: Randal L. Schwartz
Company: RockSoft
Department: R&D
Date: June 12, 2008 6:38 pm
Version: 2.1
Size: 21k
Status: Final beta
```

简言之，我们需要在 3 个地方进行修改。作者（Author）的名字需要更改，日期(Date)需要更新，电话（Phone）需要删除。我们需要在上百个这样的文件中作这些修改。

Perl 可以通过尖括号操作符(<>)对文件进行修改。下面程序能完成我们希望的工作，虽然第一次看时，不是很明显。这个

程序只有一个新的特性：特殊变量$^I；现在可以不用管它，我们将在后面讨论：

```
#! /usr/bin/perl –w

use strict;

chomp(my $date = ‘date’);
$^I = “.bak”;

while(<>){
  s/^Author:.*/Author: Randal L. Scharwartz/;
  s/^Phone:.*\n//;
  s/^Date:.*/Date: $date/;
  print;
  }
```

由于需要当前的日期，因此在程序开端使用了系统命令：date。另一种获得时间的更好的方法是（格式有些不同）使用 Perl 自带的 localtime 函数，其在标量 context 中使用：

```
my $date = localtime;
```

下一行是给$^I 赋值，我们现在不讨论它。

根据我们现在所知的，上述操作的结果是文件中新修改的部分被输出到终端，内容快速滚动，但文件本身不会被修改。

由尖括号操作（<>）所得到的文件列表来源于命令行。主循环读入，更新，输出每一行。（根据我们现在所知的，上述操作的结果是文件中被修改的部分被输出到终端，这些内容快速滚动，但文件本身不会被修改。）第二个替换操作将含有电话(phone)号码的整行由空串替换，连换行符一起替换掉。这行输出时，什么也不会出现，就像电话(Phone)号码从没存在过一样。还有大量的行不会被这三个模式所匹配，他们在输出时不会有任何更改。

结果接近于我们所期望的了，除了还不知道怎样将更新的信息写回文件。答案是变量$^I。默认时为 undef，此时没有什么特殊的地方。但给它设置某些串时，它使尖括号操作(<>)变得有些特殊。

我们知道尖括号(<>)的神奇特点：如果没有指定文件名，则其从标准输入流中自动打开和关闭一系列文件进行读入。但如果$^I 中有字符串，这个字符串则会成为备份文件的扩展名。我们在下面仔细讨论。

我们假设此时尖括号(<>)打开的文件是 fred03.dat。它像以前那样打开它，但进行了重名名，把它叫做 fred03.dat.bak◆。这很好，因为不在使用之前的名字。现在<>将这个新的文件作为默认的输出，因此任何内容将输出到那个文件中◆。while 循环会从旧的文件中读入一行，更新它，再把它输出到新文件中。在一台普通的机器上运行这个程序，几秒钟就能更新上百个文件。非常强大，不是吗?

◆过程的细节之处，在不同的 non-Unix 可能有所不同；但结果几乎是一样的。阅读移植到你的系统中 Perl 的发布须知(release notes)。

◆<>尽量继承以前文件的权限和所有者等设置；例如，如果以前的文件是全局可读的(world-readable)，则新文件也应当是全局可读的。

当程序结束时，用户能见到什么？用户说，“啊，我看到了发生的改变。Perl 修改我的文件 fred03.dat，做了我希望的修改，并将早期的文件保存在叫做 fred03.dat.bak 的文件之中”但我们知道的真相是：Perl 不会修改任何文件。它新建了一份修改后的拷贝，说“Abracadabra（咒语）”，当在魔术棒出现过闪光后，文件就被交换了。很狡猾吧！

某些人喜欢将~ (tilde）作为$^I 的值，这同 emacs（Unix 上的一个编辑器）处理备份文件类似。$^I 另一个可能值是空串，这会修改文件，但不会将原来的文件进行备份。由于在模式中少输入几个字符就能将以前的数据去掉，因此除非有特殊的理由，否则不要使用空串。毕竟当这一切完成时，再删除备份文件也是很容易的。如果什么地方出了问题，可以将备份文件命名成以前的文件，知道如何使用 Perl，你将非常轻松（参看第十三章多个文件重命名的例子）。

9．5．4 在命令行中进行修改

前一节的例子是非常容易写的。但 Larry 觉得它还不是特别的简单。

想象你需要更新上百个文件，这些文件中 Randal 拼写错误，多写了一个 l：Randall。你可以写一个类似于前面的程序。或者在命令行中使用下面的一行程序：

```
$perl –p –i.bak –w –e ‘s/Randall/Randal/g’ fred*.dat
```

Perl 有完整的命令行选项，通过它们只需在命令行中输入几个字符，就能构建一个完整的程序◆。我们来分析上述程序做了些什么。

◆查看 perlrun 了解完整的选项列表。

开始的命令 perl 同文件顶端的 #! /usr/bin/perl 一样：它是指使用 perl 程序来处理后面的部分。

-p 要求 Perl 为你写一个程序。它算不上是一个完整的程序；看起来有些像下面的◆：

◆print 出现在 continue 块中。查看 perlsyn 和 perlrun 的帮助手册了解更多信息。

```
while(<>){
  print;
}
```

如果想更少，可以使用 –n 替代；它会自动省略掉 print 语句，你可以只输出你感兴趣的部分。（ awk 的粉丝(fans)知道 –p 和 –n。）再声明一次，它算不上是一个完整的程序，但能省略掉大量的输入。

下一个选项是 –i.bak，它在程序开始处将$^ I 的值设置为“.bak”。如果不需要备份文件，可以使用 -i 参数，后不接任何的扩展名。如果不需要多余的降落伞，则可只在飞机上留下一个。(if you don't want a spare parachute, you can leave the airplane with just one.)（这里仅是一个比喻，译者注）

我们已经见过 –w：作用是将警告打开。

-e 选项涵义是“执行下面的代码。”即是说 s/Randall/Randal/g 字符串被当作 Perl 代码执行。由于我们有 while 循环（-p 选项），这段代码被放在循环内，print 之前。由于技术上的理由，-e 代码中最后一个分号是可省略的。如果有多个-e 选项，因此有多块代码，只有最后一个分号是可省略的。

最后一个参数是 fred*.dat，它指定@ARGV 应当包含匹配上此模式的文件名。这几部分综合起来，就是一个大致如下的程序，输入为匹配上 fred*.dat 的文件：

```
#! /usr/bin/perl –w

$^I = “.bak”;
while(<>){
 s/Randall/Randal/g;
 print;
}
```

将这个程序和前面的程序进行比较，可以发现它们是非常相似。这些命令行的选项是非常方便的，对吧？

9．5．5 非捕捉用的括号

现在，你已经知道括号可以捕捉匹配上的字符串，并将它们存入变量之中，如果只想用括号将某部分进行分组？考虑这样的正则表达式：只希望其中一部分括号中所匹配的内容被存入内存变量中。在下面的例子中，我们希望“bronto”是可选择的，为了将它变成可选择的，我们需要将它用括号括起来。接着，模式使用了一个模式可以得到“steak”或者“burger”，匹配上的字符串被存入内存变量中。

```
if(/(bronto)?saurus (steak|burger)/)
  {
    print “Fred wants a $2\n”;
  }
```

即便“bronto”没有被匹配上，此部分仍然会存入$1。Perl 统计开括号的个数，从而给这些变量命名。我们需要的部分被存入$2。当模式变复杂时，情况就变得非常复杂。

幸运的是，Perl 的正则表达式有一种方法可以使括号只进行分组，而不会引起内存变量的分配。我们将它叫做非捕捉用的括号（non-capturing parentheses），对于它，有一个特殊的写法。我们在开括号后面加上一个问号和冒号，(?:)◆，其作用事告诉 Perl 括号只是分组的作用。

◆这是?号在正则表达式中的第四种用法：问号，表示 0 或 1 的数量词，非贪婪修饰符，现在是开头符

改变上述正则表达式，使之对“bronto”是非捕捉用的括号，我们需要的部分被存入变量$1。

```
if(/(?:bronto)?saurus (steak|burger)/)
  {
```

```
    print "Fred wants a $1\n";
  }
```

如果以后需要改变正则表达式，如在 brontosaurs burger 上再加入 barbecue，我们可以加入"BBQ "（含有空格），并且使括号是非捕捉用的，那么我们需要的部分所对应的内存变量仍为$1。否则，可能每一次在正则表达式中加入括号时，需要改变内存变量名。

```
if (/(?:bronto)?saurus (?:BBQ )?(steak|burger)/)
  {
    print "Fred wants a $1\n";
  }
```

Perl 的正则表达式的括号还有些其它的特殊用法，它们可以完成某些复杂的功能，如向前找，向后找，内嵌注释，甚至在模式中执行代码。你可以参阅 perlre 的帮助手册了解更详细的信息。

9．6 练习

答案请参照附录 A：

1. [7]写一个模式，它能匹配$what 当前的内容的 3 份连续拷贝。也就是说，如果$what 为 fred，则此模式能匹配 fredfredfred。如果$what 为 fred|barney，则此模式能匹配 fredfredbarney, barneyfredfred, barneybarneybarney,或者其它的变种。（提示：你应当在程序的顶端设置$what 的值，如 my $what = 'fred|barney';）

2. [12]写一个程序，它可以得到当前文本文件的一个拷贝。在拷贝的文件中，字符串 Fred(大小写无关)将被 Larry 替换掉。（因此，"Manfred Mann"将变成 "ManLarry Mann".）输入的文件名已经在命令行中指定（不需要询问用户），输出的文件名是对应的输入文件名后面加上.out。

3. [8]修改上面程序，使之将 Fred 由 Wilma 替换，Wilma 由 Fred 替换。如果输入的为 fred&wilma，则输出为 Wilma$Fred。

4. [10] 额外练习：写一个程序在你所有的练习的答案前加上下面这样一行：

   a) ## Copyright (C)   20XX by Yours Truly

   将上面一行放在"shebang"行（Perl 程序的第一行，#!/usr/bin/perl（可能随 Perl 安装的位置而有所不同，但是指第一行，译者注））下面。你应当在"源文件"中修改，但请备份文件。假定你可以在命令行中同时输入程序和需要的修改的文件名。

5. [15]额外练习：修改第四题程序，如果程序已经有 copyright 这一行，则不进行修改。提示：由<>读入的文件名可以在$ARGV 中找到。

第十章 更多控制结构

在本章中，你可以见到其它书写 Perl 代码的方法。对于大多数而言，这些技术不能使 Perl 更强大，但它们能让你更容易的完成任务。你不需要在你自己的代码中使用这些技术，但请不要跳过本章。你肯定会或早或迟的在别人的代码中读到这些控制结构。很可能，在你读完本书时，就能见到它们。

10．1unless 控制结构

在 if 控制结构中，只有条件为真时，才执行块中的代码。如果你想在条件为假时执行，可以使用 unless：

```
unless($fred =~ /^[A-Z_]\w*$/i){
   print “The value of \$fred doesn’t look like a Perl identifier name.\n”;
}
```

unless 的含义是：除非条件为真，否则执行块中的代码。这和在 if 语句的条件表达式前面加上!(取反)所得结果是一样的。另一种观点是，可以认为它自身含有 else 语句。如果不太明白 unless 语句，你可以把它用 if 语句来重写（头脑中，或者实际的重写）：

```
if($fred =~ /^[A-Z_]\w*$/i){
   #什么也不做
   }else{
      print “The value of \$fred doesn’t look like a Perl identifier name.\n”;
   }
```

重写后的效率不会变化，它们会被编译成相同的内部字节码。另一种重写方法是，在条件表达式前使用取反符号(!)：

```
if(! ($fred =~ /^[A-Z_]\w*$/i)){
  print “The value of \$fred doesn’t look like a Perl identifier name.\n”;
}
```

你应当选择最容易理解的方式来书写代码，因为很可能你得维护人员也最熟悉这种写法。如果在 if 语句前加上取反符号(!)最容易理解，那就用这种方法。当然，你也可能认为，用 unless 更自然。

10．1．1unless 和 else 语句一起使用

unless 中也可以有 else 语句。虽然语法上支持，但可能引起混淆：

```
unless ($mon =~ /^Feb/){
    print “This month has at least thirty days.\n”;
  }else{
    print “Do you see what’s going on here?\n”;
  }
```

某些人可能喜欢这种用法，特别是第一部分代码很短（如只有一行代码）而第二部分较长时。但我们倾向于使用 if 语句，将条件部分取反，或者改变两部分的位置：

```
if($mon =~ /^Feb/){
  print “Do you see what’s going on here?\n”;
}else{
  print “This month has at least thirty days.\n”;
}
```

记住你的代码要给两个的“读者”看：执行代码的计算机，以及确保代码执行的人。如果他不能很好理解你的代码，那很可能计算机也不能正确执行。

10．2until 控制结构

有时，希望将 while 循环的条件部分取反。此时，可以使用 until：

```
until($j > $i){
  $j *=2;
}
```

这个循环一直执行，直到条件表达式的返回值为真为止。它和 while 循环非常类似，只是在条件为假时重复执行，而不是在条件为真的情况下执行。条件表达式在第一次迭代前即会被求值，因此这是 0 次或多次的循环，和 while 循环一样◆。同 if 和 unless 一样，将条件部分取反，即可将 until 循环写成同 while 循环一样。通常，你会发现使用 until 是非常简单和自然的。

◆Pascal 程序员应当小心：在 Pascal 中，重复的 until 语句至少要执行一次迭代，但 Perl 中的 until 循环可能一次也不执行，如果循环执行前的条件表达式的值为真。

10．3 表达式修饰符

为了得到更紧凑的形式，表达式后可以紧接控制修饰语。如，if 修饰语可以像 if 块那样使用：

```
print “$n is a negative number.\n” if $n<0;
```

上述代码和下面代码的结果相同，除了少书写括号和花括号外◆：

◆也少了些换行符。但我们应当指出花括号类型创造了新的作用域。在极少数情况下，你需要了解其完整的细节，参见相应的文档。

```
if($n < 0){
   print "$n is a negative number.\n";
}
```

Perler 一般都喜欢少输入些字符。简写的形式读起来很像英文：输出这段消息，如果$n 小于 0。

条件表达式也是先被求值的，虽然被放在后面。这和通常的的从左到右的顺序相反。要理解 Perl 代码，应当像 Perl 内部的编译器那样，将整个语句读完，来其具体的含义。

还有一些其它的修饰语：

```
&error("Invalid input") unless &valid($input);
$i *= 2 unitl $i > $j;
print " ", ($n += 2) while $n <10;
&greet($_) foreach @person;
```

它们和你预期的一样（我们希望是）。每一个都可以按前面的 if 修饰语例子类似的方法重写。例如：

```
while ($n < 10){
   print " ", ($n +=2);
}
```

应当强调下 print 语句中带括号的表达式，它将 2 加给$n，再将结果传给$n。然后，这个值被输出。

这些简写形式读起来很像自然语言：call the &greet subroutine for each @person in the list（对于列表@person 中的每一元素，调用&greet 子程序）。Double $i until it's larger than $j（将$i 加倍，直到大于$j 为止）◆。这些修饰语通常按如下语句那样使用：

◆它帮助我们那样思考。

```
print "fred is '$fred',barney is '$barney'\n"          if $I_am_curious;
```

将这些代码“反转”，可以把重要部分放在前面。上述代码的重点是观察某些变量；而非你是否好奇(you're curious.)◆。有些人习惯将这个语句写在一行里面，可能在 if 语句前都加入几个制表符(tab)，将它移到右边，如我们例子显示的那样；另一些人喜欢将 if 语句放在新行中：

◆我们自己创建的名字 $I_am_curious, 它不是 Perl 内嵌的变量。通常使用这种技术的程序员习惯将变量叫做$TRACING，或者使用 constant pragma 声明的常量。

```
print "fred is '$fred', barney is '$barney'\n"
      if $I_am_curious;
```

虽然可以将这些含有修饰语的表达式重写，但反过来不一定正确。只有单个表达式才允许按这两种方式书写。如你不能写 *something if something while something until something unless something foreach something*，这会变得极端复杂。并且也不能在修饰语左端使用多条语句。如果在修饰语两端需要使用多行语句，那按照老方法来写，使用括号和花括号。

和我们在 if 修饰语提到的那样，控制表达式(右边)首先执行，这和老方法是一样的。

对于 foreach 修饰语，不能选择其它的控制变量，它总是$_。一般情况下，这不会有什么问题，但是，如果需要不同的控制变量，你可以按传统的方法书写 foreach 循环。

10．4The Naked Block 控制结构

被称为“裸的”块是指没有关键字或条件的块。假如有一个 while 循环，大致如下：

```
while(condition){
  body;
  body;
  body;
}
```

将关键字 while 和条件表达式去掉，则有了一个“裸的”块：

```
{
  body;
  body;
  body;
}
```

“裸的”块看起来像 while 或 foreach 循环，除了它不循环外；它执行“循环体”一次，然后结束。它根本就没循环！

你也可能在其它地方使用过“裸的”块，这通常是为临时变量提供作用域：

```
{
  print “Please enter a number:”;
  chomp(my $n = <STDIN>);
  my $root = sqrt $n;            #计算平方根
  print “The square root of $n is $root.\n”;
}
```

这段代码块中，$n 和$root 这两个临时变量只在此块中有效。作为一条通用规则，所有变量应当在最小的使用范围内被声明。如果某个变量只在几行内使用，你可以将这几行放在一个“裸的”代码块中，并在其中声明变量。如果在后面还需使用$n 和$root，则需在一个更大的范围内声明它们。

你可能已经注意到了 sqrt 函数，并猜测它是什么。是的，我们以前没有介绍过它。Perl 有许多内嵌函数，要完整介绍它们，

超出了本书的范围。但如果准备好了，可以阅读 perlfunc 的帮助手册来了解更多的信息。

10．5 elsif 语句

很可能需要检测一连串的条件表达式，一个接一个，来判断哪一个是真。这可以在 if 控制结构中使用 elseif 语句做到，如下例：

```
if(!defined $dino){
   print “The value is undef.\n”;
}elsif($dino =~ /^-?\d+\.?$/){
   print “The value is an integer.\n”;
}elsif($dino =~ /^-?\d*\.\d+$/){
   print “The value is a _simple_ floating-point number.\n”;
}elsif($dino eq ‘’){
   print “The value is the empty string.\n”;
}else{
   print “The value si the string ‘$dino’.\n”;
}
```

Perl 会依次检测条件表达式。当某个成功时，相应的代码即被执行，整个控制结构也就结束了◆，然后转到后面（控制结构之后）执行。如果没有不成功，则 else 代码块被执行。（else 语句是可选的。当然，本例中包含它是非常明智的）

◆Perl 没有像 C 语言“switch”结构中的“fall-through (直接跳入)”下一个代码块的功能。

elsif 语句的个数是没有限制的，但是，如果 Perl 执行第 100 个 elsif 语句，则需执行前 99 个条件判断语句。如果需要半打(6)以上的 elsif 语句时，应当考虑是否存在更有效率的实现方法。Perl FAQ(参见 perlfaq 的帮助手册)中有许多关于如何模拟其它语言中的“case”或“switch”语句的方法。

你可能注意到关键字：elsif，只有一个 e。如果写成“elseif”，Perl 会提示拼写错误。为什么？因为 Larry 认为应该这样◆。

◆事实上，他拒绝任何建议：例如，至少应当允许它是一种合理的选择方式。“如果想写第二个 e，很简单。第一步：你自己发明一种语言。第二步：使它流行起来。”当创建你自己的语言时，你可以随意命名这些关键字。我们不希望你的语言是第一个出现像“elseunless”这样的关键字的语言。

10．6 自增和自减

常常需要一个标量变量可以自动增 1，或者减 1。由于这种操作很平常，因此同其它操作频繁的表达式一样，Perl 也提供了一种简写形式。

自增运算符(++)会使标量变量自动加 1，这和 C 以及类似语言是一样的：

```
my $bedrock = 42;
```

```
$bedrock++;            #$bedrock 的值加 1 ；现在为 43
```

和其它方法一样，如果变量上加 1 ，则此变量会根据需要自动被创建：

```
my @people = qw{ fred barney fred Wilma dino barney fred pebbles};
my %count;                          #新的空的 hash
$count{$_}++ foreach @people;       #根据情况创建新的 keys 和 values
```

第一次执行 foreach 循环时，$count{$_}自增 1 。此时，$count{“fred”}从 undef（由于之前 hash 中不存在）变成 1 。第二次执行 foreach 循环时，$count{“barney”}变成 1 ；接着，$count{“fred”}变成 2 。每执行一次循环，%count 中的某个元素增 1 ，或者新元素被创建。循环结束时，$count{“fred”}值为 3 。这提供了一种查看某个元素是否存在于列表之中，以及其出现次数的方法。

类似的，自减运算符(--)将标量变量的值减 1 ：

```
$bedrock--;            #$bedrock 值减 1 ；现在为 42
```

10．6．1 自动增量的值

获得以及改变此变量的值，可在一步内完成。将++放在变量前，先将此变量增 1 ，再取其值。这是前置++：

```
my $m = 5;
my $n = ++$m;          #$m 的值增加到 6 ，将此值赋给$n
```

或者将 - 放在变量前，再取其值。这是前置--:

```
my $c = --$m;          #$m 的值减到 5 ，再将此值赋给$c
```

下面还有一些类似的方法。变量放在前面，先得到此变量的值，再进行自增或自减。这被称为后置++或后置--:

```
my $d = $m++;          #$d 得到先前的值（ 5 ），然后自增到 6
my $e = $m--;          #$d 得到先前的值（ 6 ），再自减到 5
```

上述运算同时做了两件事。取其值，并改变其值。如果运算符(++,--)在前，则先增加（或减少），然后取其值。

如果这些表达式单独出现◆，只是利用其改变值的特点，则将操作符(++,--)放在前面或者后面是没有区别的◆：

◆也就是 void context

◆了解语言是如何实现的程序员，可能猜测后置++和后置--的效率要低一些，但 Perl 不是那样的。当在 void context 中使用时，Perl 会将后置的类型进行优化，。

```
$bedrock++;          #$bedrock 值加 1
++$bedrock;          #同上；$bedrock 值加 1
```

这种操作通常和辨别 hash 中是否存在某个元素的应用结合在一起：

```
my @people = qw{ fred barney bam-bamm Wilma dino barney betty pebbles };
my %seen;

foreach (@people){
   print "I've seen you somewhere before, $_!\n";
   if $seen{$_}++;
}
```

barney 第一次出现时，$seen{$_}++的值为 false，因为此时$seen{$_}为$seen{"barney'}，其值是 undef，同时$seen{"barney"}的值增 1 。当 barney 再次出现时，$seen{"barney"}为 true，因此此消息被输出。

10．7for 控制结构

Perl 的 for 控制结构和其它语言中的 for 控制结构类似。如下：

```
for(initialization; test; increment){
   body;
   body;
}
```

对于 Perl 来讲，这个循环和 while 循环类似，看起来大致如下◆：

◆增量在一个 continue 块中进行的，但超出了本书的讨论范围。参见 perlsyn 了解详细的信息。

```
initialization;
while(test){
   body;
   body;
   increment;
}
```

for 循环的最常用用法是，进行重复的运算：

```
for($i =1; $i <=10; $i++){                #从 1 到 1 0
   print "I can count to $i;\n";
}
```

如果以前见过这种程序，那不用阅读注释你就知道第一行的含义。循环开始前，控制变量$i 值设置为 1。接着就像进入一

个 while 循环，循环条件是$i 小于等于 10。但在循环之间，控制变量会自动增 1 。

第一次进入循环时，$i 为 1 。由于它小于等于 10，则进入循环。虽然增量表达式在循环的顶端出现，但它在循环底端，输出消息后执行。现在，$i 变成了 2 ，它也小于等于 10；我们又一次进入循环，接着$i 增到 3，它仍小于等于有 10，然后重复。

接着，我们再进入循环，然后$i 的值继续增 1 （此时为 9 ）。现在$i 变成 10，它小于等于 10。我们最后一次执行循环内的语句，此时$i 的值为 10。最终，$i 增到 11，它大于 10。则循环结束，程序进入剩下的部分。

这三部分同时出现在顶端，对有经验的程序员来讲，当读到第一行程序时："啊，这是一个循环，$i 的值从 1 到 10。"

循环结束时，控制变量的值要"大" 1。在本例中，控制变量的值为 11◆。这个例子是通用的。例如，可以从 10 到 1：

```
for($i = 10; $i >=1; $i--){
  print "I can count down to $i\n";
}
```

下例从 -150 到 1000，步长为 3◆：

◆它不是精确得到 1000。最后一个数字是 999，因为$i 的值必须是 3 的倍数。

```
for($i = -150; $i <= 1000; $i +=3){
  print "$i\n";
}
```

这三个部分（初始化，条件判断，步长）的任意部分均能为空，但分号不能省略。在下面这个不常见的例子中，条件判断部分为一个替换表达式，步长部分为空：

```
for($_ = "bedrock"; s/(.)//; ){          #如果 s///成功则进入循环
  print "One character is: $1\n";
}
```

条件判断部分是一个替换表达式，它在替换成功时返回 true。在本例中，第一次进入循环，会将 bekrock 的第一个 b 去掉。每一次迭代均会移除一个字符，当字符串为空时，替换失败，循环结束。

如果条件判断表达式（分号中间那个）为空，则恒真，此时为无限循环。除非你知道怎样从这类无限循环中退出，否则不要写出这样的代码，推出的方法在本章后面讨论：

```
for(;;){
  print "It's an infinite loop!\n";
}
```

下面是更像 Perl 无限循环的例子：while 循环，如果你需要的话◆：

◆如果不小心进入无限循环，试试 Ctrl+C 能否退出。当按下 Ctrl+C 后，很可能在终端中还有许多输出信息，这依赖于你的系统 I/O，以及一些其它的因素。

```
while(1){
 print “It’s another infinite loop!\n”;
}
```

C 程序员熟悉第一种方法，但甚至初级的 Perl 程序员也知道 1 恒真，因此如果需要无限循环，通常第二种方法比第一种更好。Perl 能识别出上面那样的常数表达式，并对其优化，因此不会有效率上的问题。

10．7．1foreach 和 for 的关系

对于 Perl 解析器(parser)而言，关键字 foreach 和 for 是等价的。也就是说，当 Perl 遇见其中之一时，和遇见另一个是一样的。Perl 通过括号能理解你的目的。如果其中有两个分号，则是 for 循环（和我们刚讲的类似）；如果没有，则为 foreach 循环：

```
for(1..10){                    #实际上是 foreach 循环,从 1 到 1 0
   print “I can count to $_!\n”;
}
```

这实际是一个 foreach 循环，但写作 for。除了本例外，本书后面均写作 foreach。在实际代码中，你觉得 Perler 会输入这四个多余的字符吗◆？除了新手外，一般均写作 for，你也应当像 Perl 那样，通过查看括号内分号个数来判断。

◆如果你认为是，那你没有理解我们的观点。在程序员中，特别是 Perl 程序员，懒惰是一种美德。如果不信，你可以在下一届 Perl Mongers 集会上询问他们。

在 Perl 中，foreach 循环通常是一个更优的选择。在前面 foreach 循环（被写成了 for）的例子中，很容易看到其范围是由 1 到 10。但你能看出下例中有什么问题吗？在找到答案前不要看脚注中的答案◆：

◆这里有 2.5 个错误。条件判断部分使用小于(<)号，因此实际的循环只执行 9 次而非 10 次。第二，控制变量是$i，但循环体中使用的却是$_。还有半个错误是，对于这种写法，读，写，维护，调试的代码更多，这也是我们说 foreach 形式是一个更好选择的原因。

```
for($i = 1; $i < 10; $i++){                    #Oops!某些地方有错误。
   print “I can count to $_!\n”;
}
```

10．8 循环控制

Perl 是一种“structrued(结构)”程序语言。每一个代码块均有一个入口，即是块的顶端。但有时需要对它进行更多的控制。例如，你可能希望写一个至少执行一次的 while 循环。或者希望能更早的从代码中退出。Perl 提供了三种循环控制的操作，可在循环块中起到某些作用。

10．8．1 last 操作

last 会立刻结束循环。(这同 C 语言或其它语言中的"break"语句类似)。它从循环块中"紧急退出"。当执行到 last，循环即结束，如下例：

```
#输出所有出现 fred 的行，直到遇见 __END__标记
while(<STDIN>){
  if(/__END__/){
    #这个标记之后不会有其它输入了
    last;
  }elsif(/fred/){
    print;
  }
}
##last 跳转到这里##
```

当输入行有__END__标记时，循环即结束。最后的注释行是不需要的，我们这里写出来只是为了把问题说清楚。

Perl 的 5 种循环体分别是 for, foreach, while, until，以及"裸"块◆。花括号括起来的 if 块，子程序◆不算。在上面的例子中，last 对整个循环块其作用。

◆是的，可以使用 last 从"裸的"代码块中跳出。这和从外面跳入"裸的"块中是不同的。

◆这可能是一个坏主意，但可以在子程序里面使用循环控制操作符控制子程序外面的循环。也就是说，如果在循环块内调用一个子程序，子程序内执行 last 操作，同时子程序没有循环块，那么程序的流程会跳到主程序循环块的后面。在将来的 Perl 中，这种在子程序内的循环控制能力会被去掉，没有人会怀念它的。

last 常用在最内层的循环体中，可以从中跳到外面来；这在下面会介绍。

10．8．2 next 操作

有时还不希望结束循环，但本次循环已经结束。这种情况下，next 是非常适用的。它跳到当前循环块的最后面（块内）◆。next 之后，又会进入下一轮循环（这和 C 或者类似语言的"continue"相似）：

◆我们这里又撒了一个小谎。事实上，next 跳到循环的 conttinue（通常省略）块的开头处。参见 perlsyn 了解详细的信息。

```
#分析输入文件的单词
while(<>){
  foreach(split){                    #将$_分拆成单词，并依次赋给$_
    $total++;
    next if/\W/;                     #不是"words"的被跳过
```

```
        $valid++;
        $count{$_}++;                    #对每个单词进行计数
        ##next 跳到这里##
      }
    }
    print "total things = $total, valid words = $valid\n";
    foreach $word (sort keys %count){
      print "$word was seen $count{$word} time.\n";
    }
```

这几乎是到目前为止所出现过的最复杂例子，让我们一步一步地分析它。while 循环从尖括号输入符(<>)中读入，一行接一行的，每次都读入$_中；你以前已经见过了。循环每次执行时，这一行均被读入 $_ 中。

在循环体中，foreach 循环在 split 的返回值上进行迭代操作。你还记得没有参数的 split 的默认参数是什么吗◆？是$_，分隔符时空白（whitespace），它将$_分拆成一串单词。由于 foreach 循环没有提到其它的控制变量，那么它为默认的$_。因此，这些单词依次赋给$_。

◆如果不记得了，也不用担心。不要花费精力在能在 perldoc 能查到的东西上。

但，我们刚才不是说过，$_存储的是一行值吗？是的，在外部循环中，这是正确的。但在 foreach 循环内部，它存放的是单词。Perl 能处理$_的不同用法，这种事经常发生。

现在，在 foreach 循环内部，$_一次含有一个单词。$total 值依次增加，它统计单词的总数。接着一行（本例重点）检查是否有 nonword 字符：存在字符，数字，下划线之外的符号。因此如果它是 Tom's；或者含有逗号，引号，其它的特殊字符，则会匹配上此模式，从而跳过循环的后部分，处理下一个单词。

但如果是一个普通的单词，如 fred，$valid 值增 1，同时$count{$_}也会增 1，每一个单词均会被计数。当这两重循环结束时，我们就把用户感兴趣的文件中的所有单词进行了统计。

我们不打算解释后面几行。我们希望你现在已经能解决它们了。

last, next 可以在这 5 种循环中使用：for, foreach, while, until, 或者"裸"块。如果块是嵌套的，next 在最里层工作。在本节结束处你能了解到怎样改变它。

10．8．3 redo 操作

循环控制的第三个操作是 redo。它会调到当前循环块的顶端，不进行条件表达式判断以及接着本次循环。（在 C 或类似语言中没有这种操作。）下面是一个例子：

```
    #输入测试
    my @words = qw{ fred barney pebbles dino Wilma betty };
    my $errors = 0;
```

```
foreach(@words){
   ##redo 跳到这里##
   print "Type the word '$_': ";
   chomp(my $try = <STDIN>);
   if($try ne $_){
      print "sorry – That's not right.\n\n";
      $errors++;
      redo;                                 #跳转到循环顶端
   }
}
print "You've completed the test, with $errors errors.\n";
```

和其它两种操作一样，redo 可以在 5 种循环体内工作，当在嵌套循环体中时，它在最内层使用。

next 和 redo 的最大区别在于，next 会进入下一次循环，而 redo 会继续执行本次循环。下面这个例子能让你感受到这个 3 种操作的不同◆：

```
foreach(1..10){
   print "Iteration number $_.\n\n";
   print "Please choose: last, next, redo, or none of the above?";
   chomp(my $choice = <STDIN>);
   print "\n";
   last if $choice =~ /last/i;
   next if $choice =~ /next/i;
   redo if $choice =~ /redo/i;
   print "That was't any of the choices…onward!\n\n";
}
print "That's all, folks\n";
```

如果仅输入回车（尝试 2 或 3 次），循环会顺次执行。如果在 4 时选择 last，循环立即结束，而不会进入"5"。如果在 4 时选择 next，则进入 5，但不会打印出 "onward"的信息。如果在 4 时选择 redo，则会重复一次。

10．8．4 标签块

如果要从最内层的循环中跳出来，可以使用标签(label)。标签在 Perl 中就像一般标识符一样：由字母，数字，下划线组成，但不能由数字开头。由于没有前缀字符，标签可能和内嵌的函数名，或者你自己的子程序名混淆。如果将标签取名为 print 或 if，是非常不好的。由于这些原因，Larry 推荐标签均大写。这会防止标签和其它标识符冲突，同时也使之在代码中更突出。同时，标签很少使用，通常只在很少一部分程序中出现。

要给循环体加上标签，可在循环前面加上标签和冒号。在循环体内，可以根据需要在 last, next,或者 redo 后加上标签名，如：

```
LINE: while(<>){
      oreach (split){
```

```
        last LINE if /_ _END_ _/;    #推出 Line 循环
        …
    }
}
```

为了增加可读性，通常将标签放在左侧，不管当前的代码是否缩进了很多。标签在整块之前使用；不是针对代码中的某些点◆。在前面的代码块中，_ _END_ _表示输入结束。当这个标记出现时，程序会忽略掉剩下的行（甚至剩下的文件）。

◆因为这不是 goto。

通常给标签取一个名词性的名字更有意义◆。如上例中，由于外层循环一次处理一行，因此我们将它叫做 LINE。如果需对内层循环命名，可叫做叫做 WORD，因为一次处理一个单词。这对于说"移到下一个单词((move on to the) next WORD)"或者"重复当前行(redo(the current) Line)"是非常方便的。

◆当然，这样做总比不这样做更有意义。Perl 并不关心你将标签命名为 XYZZY 或者 PLUGH。

10．9 逻辑操作符

Perl 含有所有必须的逻辑操作符来处理 Boolean(true/false)值。例如，进行逻辑判断的逻辑与 AND(&&)和逻辑或 OR (||):

```
if($dessert{'cake'}&& $dessert{'ice cream'}){
  #两个同为真
  print "Hooray! Cake and ice cream!\n";
}elsif($dessert{'cake'} || $dessert{'ice cream'}){
  #至少一个为真
  print "That's still good…\n";
}else{
  #没有一个为真—什么也不做（我们很难过）(dessert 甜点，cake 蛋糕，ice cream 冰激凌，作者的幽默，译者注))
}
```

上面代码中的逻辑部分可能不会完全判断。如果逻辑与（&&）操作的左侧为 false，则整个为 false，因为逻辑与在两个均为 true 时其结果才为 true。在这种情况下没有理由检测右侧，因此右侧不会被求值。考虑下述例子在$hour 为 3 时会发生什么：

```
if((9 <= $hour) && ($hour < 17)){
  print "Aren't you supposed to be at work…?\n";
}
```

同样的，如果逻辑或（||）左侧结果为 true，则右侧部分也不会被求值。考虑下述例子在$name 为 fred 的情况：

```
if(($name eq 'fred') || ($name eq 'barney')){
  print "You're my kind of guy!\n";
}
```

由于这种行为，这些操作通常被称作“短路（short-circuit）”逻辑操作。它们会在可能的情况下立即返回结果。事实上，我们经常利用这种性质。假设你要计算平均值：

```
if(($n !=0) && ($total/$n < 5)){
  print "The average is below five.\n";
}
```

在这个例子中，右侧只在左边的值为 true 时才会执行，因此可以阻止像除数为 0 而引起程序崩溃这样的错误发生。

10．9．1 短路操作的值

和 C（以及类似的语言）不同的地方是，短路操作的结果是最后被执行语句的返回值，而非仅仅是一个 Boolean 值。结果是相同的。如果最后被执行的部分为真，则整个为真；为假，则整个为假。

但这是一种更有用的返回值。这些理由之一就是，逻辑或（OR）可以方便的选择一个默认值：

```
my $last_name = $last_name{$someone} || '(No last name)';
```

如果$someone 不在 hash 中，则左侧值为 undef，为 false。从而，右侧代码将被执行，并将其作为默认值◆。在后面还有利用这种特性的应用。

◆这种用法中，默认值（No last name）不仅会替换 undef 值，也会替换所有的 false 值。这对于大多数名字都没什么影响，但 0 和空串是有用但为 false 的值。这种用法应当在希望替换所有 false 值的情况下使用。

10．9．2 三元操作符 ?:

当 Larry 决定 Perl 应当具有哪些操作符时，他不希望以前的 C 程序员在 Perl 中找不到 C 中出现过的操作符，因此他在 Perl 中实现了所有 C 的操作符◆。这意味着 C 中最容易混淆的操作符：三元操作符（?:）也被移植过来了。虽然它带来了麻烦，但有时也是非常有用的。

◆坦白讲，他将那些在 Perl 中没有用的去掉了，如那些能将数字转换成变量内存地址的操作符。同时也添加了一些操作符（如字符串连接操作符），这让 C 程序员非常眼红。

三元操作有些像 if-then-else 一样，不过是在一个表达式之中。被称作“三元”操作符是因为它有三个操作数，看起来如下：

```
Express ? if_true_expr : if_false_expr
```

首先，表达式被求值。如果为 true，则第二个表达式被执行；否则，第三个表达式被执行。右侧两个表达式中有且仅有一个被执行，另一个则被忽略。也就是说，如果第一个表达式为 true，则第二个表达式被执行，第三个被忽略。如果第一个表达式为 false，则第二个被忽略，第三个被执行，并作为表达式的值。

在下例中，&is_weekend 决定将哪一个表达式的值赋给$location：

```
my $location = &is_weekend($day) ? "home" : "work";
```

下例中，如果没有平均值，则输出一些连接线（—）：

```
my $average = $n ? ($total/$n) : "-------";
print "Average: $average\n";
```

你可以将这些 ?: 操作用 if 结构来书写，但没有那么方便：

```
my $average;
if($n){
  $average = $total /$n;
}else{
  $averge = "------";
}
print "Average: $average\n";
```

这里有一个书写存在多个分支的编程技巧：

```
my $size =
   ($width < 10 ) ? "small" :
   ($width < 20) ? "medium" :
   ($width < 50) ? "large" :
                          "extra_large";    #default
```

上述代码中有三个嵌套的 ?:，当熟练掌握时，这能给你的工作带来便利。

你并非必需使用它不可。新手经常避免使用。但你可能在其他人的代码中见到其踪影，我们希望某一天你能找到一个在你自己的程序中使用它理由。

10．9．3 控制结构：使用部分求值的操作符

前面三个操作符 &&, || , ?:，均有一个共同的特殊性质：根据左侧的值（true 或 false），来判断是否执行右侧代码。有时会被执行，有时不会。由于这个理由，这些操作符有时叫做部分求值（partial-evaluation）操作符，因为有时并非所有的表达式均被执行。部分求值操作符是自动的控制结构◆。并非 Larry 想引入更多的控制结构到 Perl 中来。而是当他决定将这些部分求值操作符引进 Perl 后，它们自动成了控制结构。毕竟，任何可以激活或解除某块代码的执行即被称作控制结构。

◆某些人会猜想为什么在这一章介绍逻辑操作，你会不会呢？

幸运的是，只有利用这些表达式的副作用，如改变变量值，或引起某些输出，才会见到它们。例如，下面的代码：

```
($m < $n) && ($m = $n);
```

注意：逻辑与(&&)的结果没有赋给任何变量◆，为什么呢？

◆当然有可能为返回值，如是子程序的最后一个表达式。

如果$m 小于$n，则左侧为 true，右侧赋值表达式即被执行。如果$m 的值不小于$n，左侧为 false，则右侧被跳过。上述代码和下面的结果一样，只是这里的更容易理解：

```
if($m < $n) {$m = $n }
```

也许你在维护程序，可能遇到下面这样的代码：

```
($m > 10) || print "why it it not greater?\n";
```

如果$m 大于 10，则左侧为 true，逻辑或(OR)运算执行完毕。如果不是，则左侧为 false，右侧信息即被输出。这可以（或者说应当）按传统的方法来书写，如用 if 或 unless 等。

如果你很敏锐，那你可以像像读英文那样读这些代码。例如，检查$m 是否小于$n，如果是，则进行赋值。检查$m 是否大于 10，如果不是，则打印出消息。

通常，C 语言背景的程序员，或以前的 Perl 程序员习惯于这样书写控制结构。它们为什么要这么做呢？一些人错误地认为这样做的效率更高。另一些人认为这些技巧让他们的代码很酷。还有一些人仅仅看见其他人这么做，他们就这样做而已。

同样的，三元操作符也可以用来做控制结构。下例中，我们想将$x 赋给两个变量中较小的一个：

```
($m < $n) ? ($m = $x) : ($n = $x);
```

如果$m 更小，则得到$x。否则，$n 得到$x。

还有一种书写逻辑与（AND）与逻辑或（OR）的方法。可以使用单词：and 与 or◆。这些单词操作符和符号操作符的含义相同，但它们(单词操作符)在优先级的底端。由于它们（单词操作符）比表达式自身的结合性更低（按照优先级来讲：译者注），因此不需要括号：

◆还有低优先级的 not(同逻辑非：!)以及很少使用的 xor。

```
$m < $n and $m = $n;              #但用对应的 if 语句来书写更好
```

但最好使用括号。优先级是容易出问题的。除非能明确它们之间的优先级关系，否则最好还是使用括号。单词操作符的优先级非常低，通常可以容易的将这些表达式分成两部分来理解，先分析左侧的，再（如果需要）分析右侧的。

虽然使用逻辑运算符作为控制结构可能引起混淆，但有时也接受这种写法。在 Perl 中打开一个文件的惯用手法是：

```
open CHAPTER, $filename
   or die "Can't open '$filename':$!";
```

通过使用低优先级的短路操作符 or，我们要求 Perl “open this file…or die!”。如果打开成功，则返回 true 值，or 运算结束。如果失败，则继续执行右侧代码，返回失败的消息。

因此，使用这些操作符作为控制结构是 Perl 的惯用手法中的一种。恰当的使用它们，能增进你的代码的威力；否则将使代码难于维护。但请不要过多的使用它们◆。

◆如果一个月超过了 1 次的使用这些怪异的模式（or die 除外），即被认为是过多的使用了。

10．10 练习

答案参见附录 A：

1. [25]写一个程序，能重复要求用户猜测某个在 1 到 100 之间的数字，直到猜对为止。你的程序应当能随机的产生一个数字，使用公式 int(1 + rand 100)◆。当用户猜测错误时，程序应该回应“Too high”或者 “Too low”。如果用户输入 quit 或 exit，或者回车时，程序应立即退出。如果用户猜测正确，程序也退出。

◆如果不了解这些函数，可以参见 perlfunc 帮助手册中关于 int 和 rand 的部分。

第十一章 文件检验

早些时候，我们介绍了怎样打开一个文件句柄进行输出。通常，这会创建一个新的文件，如果存在同名的文件，则同名文件会被覆盖掉。很可能，你需要检查是否存在同名的文件。或者想知道某个文件存在多久。亦可能想知道大小大于某个数值且一定时间内没被访问的文件。Perl 提供了完备的方法来对这些文件信息进行检测。

11．1 文件检测操作

如果程序会建立新的文件，在程序创建新文件之前，我们应先确定是否存在同名的文件，以免重要数据被覆盖掉。对于这种问题，我们可以使用–e 选项，检测是否存在相同名字的文件：

```
die “Oops! A file called ‘$filename’ already exists.\n”
  if –e $filename;
```

我们在 die 消息中没有使用 $!，因为这里不是系统拒绝请求。下例检查文件是否被更新。在本例中，我们检测一个已经打开的文件句柄，而非文件名。假设程序的配置文件每周或两周就需要被更新。（可能是检查计算机病毒。）如果文件在过去 28 天内都未被修改，则什么地方出问题了：

```
warn “Config file is looking pretty old!\n”
  if –M CONFIG > 28;
```

第三个例子有些复杂。假设磁盘空间快被填满了，我们不打算购买更多的磁盘，而是决定将那些大的且没用的文件移到备份磁带上去。因此我们首先检查文件列表◆，找到那些大于 100KB 的文件。如果一个文件仅是很大，我们不一定会将其移到备份磁带上去，除非同时其在最近 90 天内都未被访问。（因此我们知道这些文件不常用）◆：

◆更可能的情况是，不是将文件名的列表读入一个数组(如我们的例子中)，而是直接使用 glob(目录句柄(directory handle))，这将在第十二章介绍。由于我们还没学习到它，因此这里就使用上述方法。

◆对于这个例子，本章结束处有一个更有效的解法。

```
my @ariginal_files = qw/ fred barney betty Wilma pebbles dino bam-bamm /;
my @big_old_files;     #要移到备份磁带上的文件
foreach my $filename (@original_files){
  push @big_old_files, $filename
    if –s $filename > 100_100 and –A $filename > 90;
}
```

这是第一次出现，你可能已注意到 foreach 循环的控制变量前使用了 my。这声明其作用域为循环内，因此本例中亦可以使

用 use strict。如果没有使用 *my* 这个关键字，那可能使用全局变量$filename。

检测文件的选项看起来是由连接线(-)和一个字母组成，字母指测试的类型，后接被测试的文件名或文件句柄。大多数选项返回 true/false 值，但有少数例外。参见表 11-1，其中包括完整的列表，下面还有一些针对特殊例子的讨论。

表 11-1 文件检测选项及其含义

检测选项	含义
-r	文件或目录对此（有效的）用户（effective user）或组是可读的
-w	文件或目录对此（有效的）用户或组是可写的
-x	文件或目录对此（有效的）用户或组是可执行的
-o	文件或目录由本（有效的）用户所有
-R	文件或目录对此用户(real user)或组是可读的
-W	文件或目录对此用户或组是可写的
-X	文件或目录对此用户或组是可执行的
-O	文件或目录由本用户所有
-e	文件或目录名存在
-z	文件存在，大小为 0（目录恒为 false）
-s	文件或目录存在，大小大于 0（值为文件的大小，单位：字节）
-f	为普通文本
-d	为目录
-l	为符号链接
-S	为 socket
-p	为管道(Entry is a named pipe(a “fifo”))
-b	为 block-special 文件（如挂载磁盘）
-c	为 character-special 文件（如 I/O 设备）
-u	setuid 的文件或目录
-g	setgid 的文件或目录
-k	File or directory has the sticky bit set
-t	文件句柄为 TTY(系统函数 isatty()的返回结果；不能对文件名使用这个测试)
-T	文件有些像“文本”文件
-B	文件有些像“二进制”文件
-M	修改的时间（单位：天）
-A	访问的时间（单位：天）
-C	索引节点修改时间（单位：天）

-r, -w, -x 以及-o 检测相应的属性对 effective user 或 group ID 是否为真，它是指实际负责运行此程序的用户◆。这些检测通过查看文件的权限位（permission bits）。如果你的系统使用访问控制列表(Access Control Lists(ACLs))，则这些测试就使用它。这些选项检测其权限，但并非说这种操作一定能进行的。例如，-w 对于 CD-ROM 上的一个文件是 true，但你并不能写入，犹如-x 对于某个空文件为 true，但其实际上是不可执行的。

◆-o 和-O 只和 user ID 有关，和 group ID 无关。

◆对于高阶学生来讲，-R, -W, -X, -O，针对 real user 或 group ID，这在程序执行了 set-ID 就非常重要。如果那样，这通常是指请求运行的用户。参看一本高级的 Unix 编程书籍中关于 set-ID 程序讨论的部分。

如果程序非空，-s 返回 true，但这是一种特殊类型的 true。其为文件的大小，单位为字节，当非 0 时被认为真。

Unix 文件系统◆包括几种类型，分别可有-f, -d, -l, -S, -p, -b, 以及-c 检测。任何一种必属于其中之一。如果有一个符号链接指向某个文件，则-f 和-l 均返回 true。因此如果想知道其是否为符号链接，应当先对其进行检测（在第十二章中有更多的关于符号链接的讨论）。

◆许多 non-Unix 文件系统也是这样，但并非每一检测项在任何系统中都能执行。例如，你的 non-Unix 系统中可能就没有 block special 文件（如挂载磁盘）。

时间检测，-M, -A, -C(均大写)，返回系统最后一次修改，访问，以及索引节点被修改到现在的天数◆。(索引节点包括除文件内容外的所有信息。查看 *stat* 系统调用的用户手册，或含有 Unix 内部细节的书籍。）这些值是为浮点类型，因此可能得到 2.00001 这样的值，如果文件是在两天又一秒前被修改。这些"天数(days)"和我们通常的计算法不同。例如，现在是 1:30am，如果你在半夜前 30 分钟对文件进行了修改，则-M 得到的值大约为 0.1，虽然是在"昨天"修改的。

◆在 non-Unix 系统中，有些可能不同，因为它们使用计时法可能和 Unix 使用的不同。例如，在某些系统，ctime 段（-C 查看的地方）是文件创建的时间（Unix 并不用这个时间）而非索引节点改变的时间。查看 perlport 的帮助手册。

当检查文件的时间时，可能得到负值如-1.2，其含义是最近一次访问时间是大约 30 小时后。此时间刻度的 0 点是程序开始运行的时间◆，因此这个值可能是指一个运行了很长时间的程序检测一个刚被访问过的文件。或者时间（有意或无意的）被设置成了将来某个时刻。

◆被存储在$^T 这个变量中，你可以更新它（使用$^T = time;），如果需要一个不同的开始时间。

-T 和-B 分别检测一个文件是文本的还是二进制的。但对文件系统了解的读者知道其中没有位（至少在 Unix-like 操作系统中）标明其为二进制还是文本文件，那 Perl 是怎样知道的呢？答案是 Perl 欺骗了我们：它打开一个文件，查看前面几千个字节，进行合理的猜测。如果没有太多的怪异符号，则其像文本。这有时会判断错误，如文本中有大量的瑞典语或法语单(词可能这些字符中的高位被设置了，有些像 ISO-8859 的变种，或者是 Unicode 编码)，这可能让 Perl 错误的将它们判断为二进制文件。这当然不算完美，如果需要将源文件和编译后的文件，抑或 HTML 文件和 PNGs 区别开，这些检测能够做到。

你可能认为-T 和-B 选项的结果是不一致的，因为一个文本文件不可能是二进制文件，反之亦然，但存在两种特殊情况，它们完全相同。如果文件不存在，或者不可读，则两个均为 false，因为其既不是文本文件也不是二进制文件。同时，一个空的文件，即可以说是空的文本文件，也可以说是空的二进制文件，此时则同为 true。

当给定的文件句柄是一个 TTY 时，-t 文件检测项返回 true，表明它是可交互的，因为它不是一个简单的文件或管道。当-t STDIN 返回 true 时，通常其含义是可以交互式的询问用户问题。如果为 false，则你的程序可能从文件或管道读入，而非键盘中。

不要担心，如果你不知道其余检测项的含义，你可能并不需要它们。如果你很想知道，可以找一本 Unix 编程的书。(在 non-Unix 系统中，这些检测项尽量模拟 Unix 系统中的返回结果，如果没有这个选项，则返回 undef。通常，你能猜测其行为。)

如果省略掉文件名或文件句柄这个参数(也就是说只有-r 或-s 这个参数)，其默认的参数是$_中的文件名◆。因此，要检测列表中哪些文件是可读的，可以如下书写：

◆-t 文件检测是一个例外，因为它对文件名是无效的（它们永远都不可能是 TTYs。）默认时，其检测 STDIN。

```
foreach(@lots_of_filenames){
   print "$_ is readable" if –r;    #同  –r $_
}
```

如果省略掉了某些参数，确保紧跟在文件检测项后面的不像通常的参数。例如，如果希望文件大小的单位是 KB，而非字节，你可能将-s 的返回值除以 1000（或 1024），如：

```
#文件名在$_中
my $size_in_K = -s /1000;      #Oops!
```

当 Perl 解析器看见正斜线（/），它并不将它看作除号。由于现在在给-s 找一个可选的操作数（参数），它看到的是某个像由正斜线作为分隔符的正则表达式的开端部分。为了避免这种混淆，在文件检测部分加上括号：

```
my $size_in_k = (-s) / 1024;    #使用默认的$_
```

明确给出被检测的文件参数将更加安全。

11．2 stat 和 lstat 函数

虽然这些检测项可以很好的给出文件或文件句柄相应参数的属性，但它们还没有包括所有的信息。例如，它们不能给出文件的连接数，或者所有者的 user ID(uid)。要得到文件的其余信息，可以使用 stat 函数，其返回 Unix 系统调用 stat 时相同的值（比你想知道的还多）◆。返回值或者是空列表，表明 stat 失败（通常是由于文件不存在）；或者是 13 个元素的列表，使用下例可以容易的说明:

◆在 non-Unix 系统中，stat 和 lstat，以及文件检测(file tests),将返回“最接近的可用值”。例如，如果系统没有 user IDs(那样，按照 Unix 的观点，系统只有一个“用户”），则 user 和 group IDs 的返回值可能是 0，就像只有一个用户：系统管理员。如果 stat 和 lstat 失败，则返回空列表。如果文件检测(file tests)失败（或者给定的系统中不可用），则返回 undef。查看 perlport 的用户手册了解最新的关于不同系统之间的区别。

```
my ($dev, $ino, $mode, $nlink, $uid, $gid, $rdev, $size, $atime, $mtime, $ctime, $blksize, $blockes)
        = stat($filename);
```

这些名字表明了 stat 返回的值的含义，在 stat(2)的用户手册中有详细描述。下面是其中一些重要部分的说明：

$dev 和 $ino

文件的设备号和索引节点号。它们组成了文件的“牌照(license plate)”。即便它有多个名字（硬连接（hard link）），设备号和索引节点号的组合仍是唯一的。

$mode

文件的权限位以及一些其它的位。如果使用过 Unix 命令 ls –l 来得到文件的详细信息，其每一行由类似于–rwxr-xr-x 开头。这些字母以及连接线（9 个）◆是指文件的权限，它们对应于$mode 中最不重要的位，在本例为 0755。其它位，除了这最低的 9 位外，是指文件的其余信息。mode 需要将它和本章后面的位操作结合起来使用。

◆上述字符串中第一个字符不是权限位。它指其类型：－是指普通文本，d 指目录，而 l 是指符号连接，还有些别的。ls 命令从其它位而非这 9 个权限位判断其类型。

$nlink

文件或目录的（硬）连接数。是指被检测项真实名字的个数。对于目录其值总是 2 或者更大的数，而对于文件（通常）是 1。我们将在第十二章中讨论建立文件的连接，到你你将更清楚。对于 ls –l，其为紧接权限位串的数字。

$uid 和 $gid

指文件所有权的 user ID 及 group ID。

$size

返回其大小，单位：字节，同 –s 文件检测项相同。

$atime, $mtime, 及 $ctime

这三个时间，它们按照系统的时间格式：32 位，表示从某个时刻到现在所经过的秒数，这个时刻是记录系统时间的一个任意值。在 Unix 和别的某些系统中，这个时刻从世界时间 1970 年第一个午夜开始，但在某些系统中，这个时刻可能不同。本章后面有更多的关于如何将时间转换为有用格式的信息。

当 stat 的参数是符号连接时，其返回的信息是此符号连接指向的实体的信息，而非符号连接本身的信息，除非此符号连接所指向的内容不能被访问。如果需要得到（几乎是没用的）符号连接本身的信息，可以使用 lstat 代替 stat（它按照相同的顺序返回同样的值）。如果其操作数不是符号连接，则 lstat 和 stat 返回的值相同。

同文件检测(file tests)一样，stat 和 lstat 的默认参数为$_，意指 stat 系统调用将针对$_所对应的文件进行操作。

11．3 localtime 函数

当有一个时间戳(timestamp)时（如 stat 中的），其格式有些像 1180630098。这对你没多少用，除非你将两个时间戳相减。你可能需要将它们转换为容易阅读的形式，如 “The May 31 09:48:18 2007”。Perl 可以在标量 context 中使用 localtime 函数做到：

```
my $timestamp = 1180630098;
my $date = localtime $timestamp;
```

在列表 context 中，localtime 返回一列值，其中某些值可能并非你所预料的：

```
my($sec, $min, $hour, $day, $mon, $year, $wday, $yday, $isdst)
    = localtime $timestamp;
```

$mon 是一个表示月份的数字，范围是 0 到 11，其在月份名字的数组中作为索引值是比较方便的。$year 是指从 1900 到现在的年份数，因此，需要加上 1900 来得到实际的年数。$wday 的值是从 0（星期天）到 6（星期六），$yday 指一年中的具体天数（从 0（1 月 1 日）到 364 或 365（12 月 31））。

还有两个相关的函数也经常用到。gmtime 函数同 localtime 一样，除了其返回的形式为是世界时间（曾经被叫做格林威治时间）。如果想从系统中得到当前的时间，可使用 time 函数。localtime 和 gmtime 在默认的情况下都使用 time 的当前值，如果没提供参数：

```
my $now = gmtime;    #得到当前的时间
```

要得到更多的处理日期和时间的信息，可以参见相应的模块信息。

11．4 位操作

如果需要对数字的位进行操作，如处理 stat 返回的 mode 位，则需要位操作符。它们对值进行二进制操作。按位与操作符（&）返回操作符左边和右边相应位操作的结果。例如，表达式 10 & 12 的值为 8。按位与操作只有在两个操作数中相应位的值均为 1 时其结果才为 1。因此 10（二进制：1010）和 12（1100）的按位与操作的结果是 8（1000，仅在两个操作数中对应得位均为 1 时结果才为 1）。参看图 11-1。

图 11-1 按位与操作

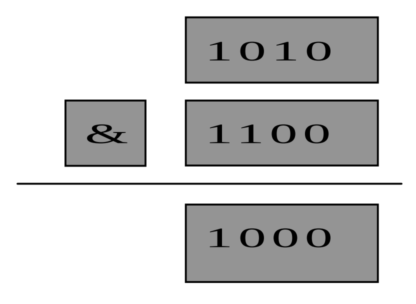


表 11-2 介绍其它位操作的含义。

表 11-2 位操作

表达式	含义
10 & 12	按位与；当操作数相应位均为 1 时结果为 1 （本例结果为 8）

10	12	按位或；当操作数中相应位有一个为 1 则结果为 1（本例结果为 14）
10 ^ 12	按位异或；当操作数中相应位有且仅有一个为 1，结果才为 1（本例结果 6）
6 << 2	位左移，将左边操作数左移右边操作数所指定的位数，被移出的位置(右边)补 0（结果为 24）
25 >> 2	位右移，将左边操作数右移动右边操作数所指定的位数，丢弃多余的位数（左边）（本例结果位 6）
~10	位取反，也叫做一元位补运算；其返回值为操作数中相应位取反的值（本例为 0Xfffffff5,这和具体情况有关）

下例对 stat 返回的$mode 进行处理。这些位处理的结果对 chmod 是有用的，chmod 将在第十二章中介绍：

```
# $mode 是对 CONFIG 进行 stat 操作所返回的 mode 值
warn "Hey, the configuration file is world-writable!\n"
  if $mode & 0002;                              #安全问题
my $classcial_mode = 0777 & $mode;              #将额外的高位屏蔽掉
my $u_plus_x = $classical_mode | 0100;          #将其中一位置 1
my $go_minus_r = $classical_mode & (~ 0044);    #将两位置 0
```

11．4．1 使用位串

所有的位操作符都可以对位串和整数进行操作。如果操作数是整数，则结果为整数。(整数至少有 32 位，可能更大，如果你的系统支持。因此，在 64 位机器上，~10 得到的是 64 位的结果：0Xfffffffffffffff5，而非 32 位的结果：0Xfffffff5。)

但如果位操作符的操作数为字符串，Perl 会将其作为位串进行处理。因此，"\xAA" | "\x55"得到的结果是"\xFF"。注意这些操作数是单字节字符串，其结果也为一个字节（8 位）。位串可以是任意长的。

这是 Perl 中少数几个需要区别字符串和数字的地方之一。参见 perlop 用户手册了解更多的关于字符串上位操作的信息。

11．5 使用特殊的下划线文件句柄

每当在程序中使用 stat, lstat，或文件检测(file test)时，Perl 就会要求系统分配一块 stat buffer 给文件 (也就是，stat 系统调用所返回的 buffer)。这就意味着如果想知道文件是否是可读且可写的，需要对相同的信息进行两次系统调用，这在系统资源充裕的条件下，不会有什么问题。

这看起来是浪费时间◆，我们可以避免它。当对 _ 这个文件句柄（此操作数为单个下划线）进行文件检测(file test)，stat，或 lstat 操作时，将要求 Perl 对前一个文件检测(file test), stat,或 lstat 函数的操作数进行操作，而非再一次的进行系统调用。有时这是危险的：一个子程序可以在你不知情的情况下调用 stat，这会将你的 buffer 覆盖掉。如果小心些，可以节约不必要的系统调用，使得程序运行更快。下例是关于将文件移到备份磁带的例子，我们使用了刚学到的技巧：

◆因为系统调用相对更慢。

```
my @original_files = qw/ fred barney betty Wilma pebbles dino bam-bamm/;
my @big_old_files;                      #需要移到备份磁带上去的文件
foreach(@orginal_files){
```

```
push @big_old_files, $_
  if (-s) > 100_100 and –A _ > 90;                    #比以前的方法效率高
```

第一次检测时使用了默认的变量$_。这也更加有效率（可能除了对程序员之外），他从操作系统中得到数据。第二次检测使用了 _ 这个文件句柄。这次检测中，使用第一次检测的文件的大小信息，这是我们所需要的。

对文件句柄 _ 进行检测不同于对$_进行检测。使用$_，可能每一次的值均不同，其为当前$_中得值，使用 _ 是为了解决重复调用系统的问题。这又是一个对完全不同的函数使用相似名字的例子。

11．6 练习

答案请参见附录 A：

1. [15]写一个程序，读入命令行中的一串文件，报告其是否可读，可写，可执行，或不存在。（提示：如果一个函数能一次对一个文件进行所有的检测将非常有帮助。）如果一个文件被执行了 chmod 0 操作，将报告什么？（在 Unix 系统中，chmod 0 some_file 将一个文件变成不可读，不可写，不可执行的）在大多数 shell 中，星号（*）表示当前目录中的所有的普通文件。也就是说，可以输入像 ./ex11-1 *这样的命令，返回当前目录下文件的属性。

2. [10]写一个程序，找出命令行中存在时间最长的文件名，并报告其天数。当参数为空时，其行为如何（例如，命令行中没有输入任何的文件）？

第十二章 目录操作

前面章节的例子中的文件和程序在同一个目录下面。但当代操作系统将我们的文件按照目录来管理，允许我们将 Beatles(披头士，著名乐队，译者注)的 MP3 和我们本书的材料存放在不同的目录中，防止我们不小心将 MP3 当作本书的材料发给出版商。Perl 允许你直接管理这些目录，这些程序在操作系统之间移植也是相当容易的。

12．1 在目录树上移动

程序在某个工作目录(working directory)下运行，这是相对路径的起点。也即是说，如果指 fred，其含义是“当前工作目录下的 fred。”

chdir 可以改变工作目录。它和 Unix shell 下的 cd 命令类似：

```
chdir “/etc” or die “cannot chdir to /etc: $!”;
```

由于这是系统请求，错误发生时将给变量$!赋值。通常应当检查$!的值，因为它将告诉你 chdir 失败的原因。

工作目录会被 Perl 启动后的所有进程所继承（我们在第十四章将更详细的讨论）。但是，对于调用 Perl 的进程的工作目录将不会改变，例如 shell◆。因此，不能写一个 Perl 程序来代替 shell 下的 cd 命令。

◆这种限制不是来源于 Perl；它是 Unix, Windows, 以及其它操作系统的一种特性。如果想改变 shell 的工作目录，参见你的 shell 的文档。

如果参数省略掉了，Perl 尝试将你的主目录(home directory)作为工作目录，这和在 shell 中使用 cd 命令不带参数时是类似的。这是少数几种在忽略掉参数而不使用变量$_的情形之一。

某些 shell 允许你在 cd 后面的参数前加上~，来使用其他用户的主目录作为当前目录（例如 cd ~merlyn）。这是 shell 的功能，而非操作系统的，而 Perl 直接调用操作系统。因此，~前缀对 chdir 无效。

12．2 Globbing

通常，shell 将每个命令行中的任何的文件名模式转换成它所匹配的文件名。这被称作 *globbing*。例如，在 echo 命令后使用了文件名模式*.pm，shell 会将它转换成它所匹配的文件名：

```
$ echo *.pm
barney.pm dino.pm fred.pm wilma.pm
$
```

echo 命令不需要知道如何展开*.pm，shell 将展开它：

```
$cat >show-args
   foreach $args (@ARGV){
      print "one arg is $arg\n";
   }
^D
$ perl show-args *.pm
one arg is barney.pm
one arg is dino.pm
one arg is fred.pm
one arg is wilma.pm
$
```

show-args 不需要知道如何进行 globbing，这些名字已经被处理后存在@ARGV 中了。

有时在 Perl 程序的内部使用像*.pm 这样的模式。我们可以轻易的将它转换为它所匹配的文件名吗？是的，只需要使用 glob 操作：

```
my @all_files = glob "*";
my @pm_files = glob "*.pm";
```

@all_files 得到了当前目录下的所有文件，这些文件按照字母排序的，不包括由点（.）开头的文件。@pm_files 和前面在命令行中使用*.pm 的例子所得到的结果相同。

任何可在命令行中是使用的，均可作为 glob 的（单个）参数，包括用空格分隔开的多个模式：

```
my @all_files_including_dot = glob ".* *";
```

这里，我们包括了额外的"点星号（.*）"参数，来得到所有的文件(由点开头的文件，以及不由点开头的文件)。引号中两个项之间的空隔是必须的◆。这在 Perl V5.6 之前的版本中能工作的原因是，glob 调用*/bin/csh*◆进行扩展的操作。由于这个原因，glob 是一个耗时的操作，例如进入大型目录或其它的情况。尽责的 Perl 黑客会避免使用 glob 来对目录进行操作，这将在后面的章节讨论。但是，如果使用最近版本的 Perl，那并不是需要关心这个问题。

◆Windows 用户习惯于使用 "*" 来表示"所有的文件"。但其含义是"所有的名字中包含点的文件"，甚至在 Windows 上的 Perl。

◆或者其等效的代替者，如果没有 C-shell 的话。

12．3 Globbing 的替换语法

虽然我们任意的使用 globbing 这个术语，我们也谈论 glob 操作，但在许多使用 globbing 的程序中并没有出现 glob 这个字眼。为什么呢？原因是，许多这类代码在 glob 操作被命名前就写好了。它使用一对尖括号（<>），和从文件句柄读入操作类似：

```
my @all_files = <*>;     ##基本上同@all_files = glob "*"一样；
```

尖括号中的值同双引号中的值一样，会被内插。其含义是指在 glob 之前，Perl 变量会被其当前值所替换：

```
my $dir = "/etc";
my @dir_files = <$dir/* $dir/.*>;
```

这里，我们从指定的目录中得到所有的文件名(有点的和没点的文件)，因为$dir 被转换成了当前的值。

由于尖括号的含义可以指从文件句柄读入或 globbing，Perl 怎么判断使用哪一个操作呢？如果尖括号之间是一个严格意义上的标识符，则其为文件句柄读入操作；否则，为 globbing 操作。如下例：

```
my @files = <FRED/*>;       ##glob
my @lines = <FRED>;         ##文件句柄读入
my $name = "FRED";
my @files = <$name/*>       ##glob
```

一个例外情况是，如果尖括号中是一个简单的标量变量（不是 hash 或数组中的一个元素），它就是一个间接的文件句柄读入操作（*indirect filehandle read*）◆，变量的内容给出了要读入的文件句柄的名字：

◆如果间接文件句柄(indirect handle)是一个文本串，那就属于"符号引用(symbolic reference)"，在 use strict 下，这种操作会被禁止。当然，间接文件句柄(indirect handle)也可能是 tyepglob 或 I/O 对象的引用(reference)，此时在 use strict 下是可以的。

```
my $name = "FRED";
my @lines = <$name>;   ##间接的文件句柄读入操作
```

判断其为 glob 或文件句柄操作是在编译时完成的，因此其独立于变量的具体内容。

如果需要，可以使用 readline 得到间接文件句柄读入的操作◆，让其看起来更加清晰：

◆如果使用的 Perl 的版本是 5.005 之后的。

```
my $name = "FRED";
my @lines = readline FRED;          #从 FRED 读入
my @lines = readline $name;         #从 FRED 读入
```

但 readline 操作并不常用，因为间接文件句柄读入操作并不常见，且通常只是针对简单的标量变量进行。

12．4 目录句柄

从给定目录得到其文件名列表的方法还可以使用目录句柄（*directory handle*）。目录句柄外形及其行为都很像文件句柄。打开（使用 opendir 而非 open），从中读入（使用 readdir 而非 readline），关闭（使用 closedir 而非 close）。不是读入文件的内容，而是将一个目录中的文件名（以及一些其它东西）读入，如下例：

```
my $dir_to_process = "/etc";
```

```
opendir DH, $dir_to_process or die "Cannot open $dir_to_process: $!";
 foreach $file(readdir DH) {
    print "one file in $dir_to_process is $file\n";
}
closedir DH;
```

同文件句柄一样，目录句柄会在程序结束时自动关闭；或者当目录句柄被重新打开而指向另一个目录时，也会自动关闭。

在 Perl 以前的版本中，globbing 它会调用别的进程；目录句柄不会，因此它更有效率，能更有效的利用系统资源。但是，这是一种底层（lower-level）的操作，意指我们需要自己处理更多的事情。

例如，返回的文件名没有特定的顺序◆。返回列表包括所有的文件，不仅仅是匹配上某个特定模式的（如 globbing 例子中的*.pm）的文件名。这些列表含有所有的文件名，包括点（.）文件，以及由点(.)开头，或点点(..)开头的文件名◆。因此，如果只想要以 *pm* 结尾的文件，我们可以在循环内部使用一个选择函数：

◆这是没有排序的结果，类似于使用 ls –f 或 find 得到的顺序。

◆不要像以前许多的 Unix 程序那样，错误的嘉定，点（.）文件以及点点（..）文件是最先返回的两个值（排序或没排序）。如果你不以前不知道，那忘掉我们刚才说的，因为这个假定是错误的。事实上，我们已经后悔在这里提到它了。

```
while ($name = readdir DIR) {
    next unless $name =~ /\.pm$/;
    … more processing…
}
```

上述的语法是正则表达式而非 glob。如果想要所有的非点（non-dot）文件(不是由点开头的文件)，可以使用：

```
next if $name =~ /^\./;
```

如果想要除了通常的点（当前目录）以及点点（父目录）之外的所有文件，我们可以明确地使用下面的语句：

```
next if $name eq "." or $name eq ".." ;
```

现在我们讨论最容易混淆的部分，请打起精神来。readdir 操作返回的文件名没有路径名部分，而只是文件名。因此，我们的得到的不是*/etc/passwd* 而是 *passwd*。由于这又是一个不同于 globbing 的地方，人们也经常在这里出现混淆。

你需要将路径名加上，以得到文件的全名（路径名+文件名）：

```
opendir SOMEDIR, $dirname or die "Cannot open $dirname: $!";
while (my $name = readdir SOMEDIR) {
      next if $name =~ /^\./;                    #跳过点文件
      $name = "$dirname/$name";                  #加上目录名
      next unless –f $name and –r name;          #除非是可读文件
```

如果没有加上目录名，则测试部分将只检测当前目录下的文件，而不是$dirname 下的文件。这是使用目录句柄最常犯的一个错误。

12．5 递归的目录列表

你在最初的 Perl 生涯中可能并不需要递归目录操作。我们不是将丑陋的 *find* Perl 脚本呈现在你面前，而是告诉你 Perl 有一个叫做 File::Find 的库，通过它你可以对递归的目录进行处理。我们这里告诉你的原因是，不希望你自己书写这样的一个程序，因为许多 Perl 程序员有了几十个小时的经验后，就喜欢进行这一类的处理，然后陷入困境，困惑于“local directory handles”以及“how do I change my directory back?”等。

12．6 操作文件和目录

Perl 通常用来处理文件和目录。由于 Perl 产生于 Unix，如今它的大多数应用仍在这上面，因此本章许多描述看起来都是以 Unix 为中心的。但一个好消息是，只要有可能，Perl 在 non-Unix 系统上几乎有相同的行为。

12．7 删除文件

许多时候，我们创建文件来让数据能保留一段时间。但是，如果文件不再需要时，则需要将它删除。在 Unix shell 中，我们可以使用 rm 命令将单个文件或一批文件删除：

```
$ rm slate bedrock lava
```

在 Perl 中，我们使用 unlink：

```
unlink “slate”, “bedrock”, “lava”;
```

这个操作将这三个文件送入了比特天堂（意指将它们删掉了，译者注），它们将不会被看见了。

由于 unlink 以一个列表作为操作，而 glob 返回列表，因此可以将它们结合起来，一次删掉多个文件：

```
unlink glob “*.o”;
```

这类似于 shell 中的 rm *.o 命令，除了我们不需要启动一个 rm 进程。因此，可以更快速的删除这些文件。

unlink 返回值告诉我们成功删除的文件数。回到第一个例子，我们可以检验其是否成功：

```
my $successful = unlink “slate”, “bedrock”, “lava”;
print “I deleted $successful file(s) just now\n”;
```

当然，如果数字为 3，则它将所有的文件均删除了，如果为 0，则一个也没删除。如果为 1 和 2 呢？那没有线索表明那些文件删除了。如果想知道，你可以使用循环，一次针对一个文件来处理：

```
foreach my $file (qw(slate bedrock lava)) {
   unlink $file or warn “failed on $file: $!\n”;
}
```

由于一次只删除一个文件，则其返回值为 0（失败）或 1（成功），它看起来像一个 Boolean 值，控制着是否执行 warn。or warn 类似于 or die，除了其不是 fatal 外（参见第五章）。这里，在 warn 的消息后加入了换行符，因为这不是我们程序的 bug 所引起的。

当某个特定的 unlink 失败时，$!变量会被设成相应的操作系统错误，我们将它包含在消息之中。这只在一次针对一个文件名进行处理时才有效，因为下一个操作系统错误会重置它。你不能使用 unlink 来删除一个目录（同不能简单的使用 rm 来删除一个目录一样）。如果要那样做，可以使用即将介绍的 rmdir 函数。

这里有一个鲜为人知的 Unix 事实。你可以有一个文件不可读，不可写，不可执行，甚至你并不拥有它，例如属于别人的，但你仍然可以删除它。这时因为 unlink 一个文件的权限同文件本身的权限是没有关系的；它和目录所包含的文件权限有关。

我们提到这一点的原因是，许多初级的 Perl 程序员，在实验 unlink 的课程中，创建一个文件，使用 chmod 将其设为 0（因此既不可读，也不可写），来检查 unlink 是否失败。事实上它一点都不抱怨的就消失了◆。如果你想 unlink 失败，试试删除 */etc/passwd* 或者类似的系统文件。由于这是一个由系统管理员控制的文件，你没有能力删除它◆。

◆某些人知道 rm 通常会提示你是否删除这一类的文件。但 rm 是一个命令，而 unlink 是系统调用。系统调用从不询问权限，它们也从不道歉说自己错了。

◆当然，如果笨到以系统管理员的身份登陆上去进行这类操作，那能得到你想要的结果。

12．8 重命名文件

将一个给定文件重命名可以很简单的使用 rename 函数做到：

```
rename “old”, “new”;
```

这类似于 Unix 的 mv 命令，将一个叫做 *old* 的文件重命名为 *new*，且在同一个目录中。甚至可以在不同的目录之间操作：

```
rename “over_there/some/place/some_file”, “some_file”;
```

这将另一个目录下叫做 some_file 的文件移到当前的目录，确保运行程序的用户有适当的权限◆。同许多请求系统调用的函数一样，rename 返回 false 如果失败，可以设置$!来得到操作系统错误，因此可以（通常因该说是应该）使用 or die(或者 or warn)来将这些报告给用户。

◆这些文件必须在同一个文件系统中。在本章稍后的地方将介绍为什么会有这条限制。

一个在 Unix shell 用法的新闻组中频繁◆出现的问题是，怎样将所有的以*.old* 结尾的文件重名命名为以 *.new* 结尾的文件。下面是 Perl 的做法：

◆这不是唯一的一个经常询问的问题；怎样一次将一批文件重命名是在这些新闻组中最常问的。这也是在大多数新闻组的 FAQ 中，这通常是第一个回答的问题。直到现在，它仍然占据第一的位置。

```
foreach my $file (glob "*.old"){
   my $newfile = $file;
   $newfile =~ s/\ .old$/.new/;
   if (-e $newfile){
      warn "can't rename $file to $newfile: $newfile exists\n";
   }elsif(rename $file, $newfile){
      ##成功，什么也不做
   } else {
      warn "rename $file to $newfile failed: $!\n";
   }
}
```

检测是否存在$newfile 的文件是需要的，因为 rename 将会立刻重命名某个文件，无论是否存在同名的文件，假定用户有权限将以前的文件删除。我们在这里进行检测的目的，就是减少这类情况所引起的信息丢失。如果想要替换一个已经存在的文件，如 *wilma.new*，则不需要使用-e 检测。

循环中的前两句可以结合（通常这样）：

```
(my $newfile = $file) =~ s/\ .old/.new/;
```

这句申明了$newfile,将$file 的值赋给它作为初始值，并对$newfile 进行更改。你可以将它读作"将$file 用右边的方法进行替换，将其结果赋给$newfile。"是的，由于优先级的关系，括号是必须的。

某些程序员第一次看到替换时会猜想为什么左边需要反斜线（\）而右边不需要。这两部分不是对称的：替换的左边部分是一个正则表达式，而右边是一个双引号括起来的字符串。因此我们使用模式/\.old$/表示"以*.old* 结尾的字符串"（是结尾的，因为我们不希望在第一次匹配时就进行替换：*betty.old.old*），而右边，我们可以只用*.new* 进行替换。

12．9 连接和文件

如果知道 Unix 的文件和目录模型，将有助于你了解更多的文件和目录的知识，无论你的 non-Unix 系统是否按这种方式工作。和以前一样，我们这里只是介绍其中的一部分，想要了解更多的信息，可以参照介绍 Unix 内部细节的优秀书籍。

挂载卷（*mounted volume*）是一个硬盘驱动器（或者任何其它类似的，如磁盘分区，软盘，CD-ROM，或 DVD-ROM）。它可能包含任意数量的文件和目录。每一个文件被存入一个被编号的索引节点(*inode*)里，可以把它想象成磁盘上的一个块。一个文件可能存放在 inode 613，而另一个可存放在 inode 7033。

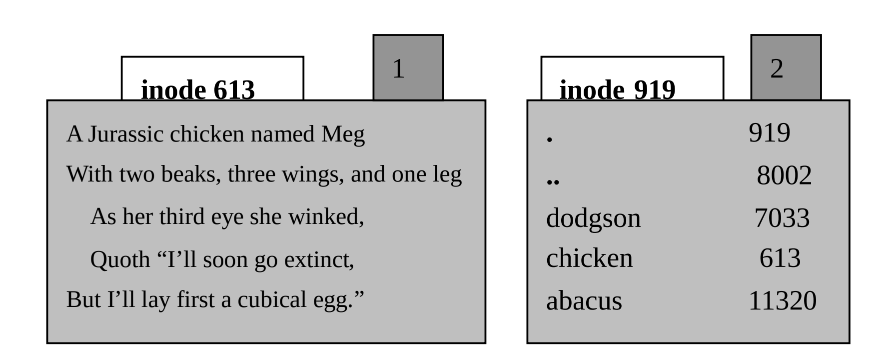
要定位某个文件，需要在目录中查找它。目录是一种特殊类型的文件，由系统维护。本质上，它是一个文件名字以及其索

引节点号的表◆。目录中除开其它的，有两个特殊的目录。一个是 . (叫作"点")，它是当前目录的名字；还有一个 .. ("点点")，它是当前目录的上一级目录（也就是，当前目录的父目录）◆。图 12-1 提供了一个描述了两个索引节点的图。一个是叫做 *chichen* 的文件，另一个是 Barneys's 的目录 poems，*/home/barney/poems*，它包含 *chicken* 这个文件。文件存在索引节点 613 中，而目录在索引节点 919 中。(目录自身的名字，*poems*，没有在图中出现，因为它被存放在另一个目录中。) 目录有三个文件（包括 *chicken*）和两个目录（其中之一是当前目录，索引节点为 919），以及每一个相应的索引节点号。

◆在 Unix 系统中（其它系统中通常没有索引节点(inode)，硬连接(hard link)，等)。可以在 ls 命令后使用-i 参数来查看文件的索引节点号。试一下命令如 ls –ail。如果给定的文件系统中多个项中有两个或多个索引节点号相同，那么实际上只有一个文件，也就是说硬盘中只有一份文件。

◆Unix 系统的根目录没有父目录。在这个目录里，.. 同目录 . 是一样的，他们均指系统的根目录。

图 12-1. 蛋之前的鸡(The chicken before the egg)



当在给定的目录中添加新文件时，系统中新增一个条目，包含文件名以及新的索引节点。系统怎么知道某个索引节点是否有效呢？每一个索引节点包含一个数字，被叫做连接数（*link count*)。如果没有在任何目录中出现，则连接数为 0，因此如果索引节点的连接数为 0，则新文件可以使用它。当索引节点被添加到目录中，则连接数增加；当从列表中删除时，连接数减少。对于刚才介绍的 *chicken* 这个文件，连接数 1 在索引节点数据的右上面。

但某些索引节点存在不同点。例如，我们已经知道每一个目录都包括 . ，而它指向自身的索引节点。因此目录的连接数至少为 2：一次出现在父目录的列表中，一次出现在自身的列表中。另外，如果它有子目录，则每一个将增加一个连接数，因为这些子目录都会包含..◆。在图 12-1 中，目录的索引节点 2 出现在数据的右上方。连接数是这个索引节点的真实名字的个数◆。一个普通文件的索引节点在一个目录中可以出现多次吗？当然可以。假设在上述的目录中，Barney 使用了 Perl 的 link 函数创建了一个新的连接：

◆这暗示了一个目录的连接数等于 2 加上其包含的子目录的个数。在某些系统中这是对的，但也有些系统可能有不同的处理方式。

◆一般 ls –l 的输出中，硬连接数出现在权限位(如 –rwxr- xr-x) 的右边。现在你知道为什么目录的值大于 1 而普通文件通常为 1 的原因了。

```
link "chicken", "egg"
```

```
or warn “Can’t link chicken to egg: $!”;
```

这和在 Unix 的 shell 提示符后输入 ln chicken egg 的结果是差不多的。如果 link 成功，则返回 true。如果失败，则返回 false，并将失败的原因赋给$!，巴尼(Barney)就能从中查看失败的原因。当运行完上述程序后，*egg* 是文件 *chicken* 的别名，反过来说也对；没有一个名字比另一个更真，并且（你可能猜想）需要花些功夫才能知道哪一个先出现。图 12-2 演示了这种新情况，其中有两个连接到索引节点 613。

图 12-2 *egg* 连接到 *chicken*

inode 613 2

A Jurassic chicken named Meg
With two beaks, three wings, and one leg
As her third eye she winked,
Quoth ‘I’ll soon go extinct,
But I’ll lay first a cubical egg.”

inode 919 2

.	919
..	8002
dodgson	7033
chicken	613
abacus	11320
egg	613

这两个文件名均指向磁盘中的同一块地方。如果文件 *chicken* 包含 200 字节的数据，则 *egg* 也含有相同的 200 字节数据，总共也只有 200 字节的数据（因为它们是同一个文件的不同名字）。如果巴尼在文件 *egg* 后添加了一行新的文本，则这一行也会出现在 *chicken* 中◆。如果巴尼不小心（或者故意的）删除了 *chicken*，其数据也不会丢失，因为仍然可以通过 *egg* 得到。其中也包含着 *chicken* 的内容。当然，如果把它们两个都删除了，则数据将丢失◆。这里还有一条关于目录中连接的规则：给定目录中的索引节点数同其挂载卷(mounted volume)上的索引节点数是一致的◆。这条规则保证了，如果物理介质（可能是磁盘）被移植到另一台机器上时，所有的目录和文件仍在相应的地方。这也是你可以使用 rename 将一个目录下的文件移到另一个目录，但这两个目录必须是在同一个文件系统中(挂载卷上)。如果它们在不同的磁盘上，系统需要重新安置索引节点的数据，这种操作对于一个简单的系统调用来讲太复杂了。

◆如果想试验创建连接和改变文件内容，需要注意的是许多文本编辑器不会直接修改原文件并保存，而是将其保存在一个修改的副本之中。如果巴尼打算用文本编辑器修改 *egg*, 则很可能得到一个新文件 *egg*，以及 *chicken*，两个不同的文件，而不是同一个文件的两个连接。

◆虽然系统也许不会立刻重写这个索引节点，但当连接数变成 0 时，没有简单的方法来恢复其数据。你最近做了备份吗？

◆一个例外是特殊的 ..，卷的根目录，它指向安装此卷的目录。

连接的另一个限制是不能给目录创建新的名字，因为目录是按照层级的方式安排的。如果你可以改变它，工具程序如 *find* 和 *pwd* 在文件系统中将迷失方向。

因此，不能针对目录使用连接，它们也不能在不同的挂载卷之间使用。幸运的是，有一种新方法能打破连接的这种限制—使用一种不同的连接：符号连接（*symbolic link*）◆。符号连接（也被叫做软连接(*soft link*)，以区别于真或硬连接（*hard links*））是目录中的一种特殊实体，告诉系统到其它的地方去找。让我们假设巴尼(在前面的 poems 目录下操作)使用 Perl 的 syslink

函数创建了一个符号连接，如下：

◆某些老式的 Unix 系统不支持符号连接(sysmlinks)，但这些系统如今已不常见。

```
symlink "dodgson", "carroll"
  or warn "Can't sysmlink Dodgson to carroll: $!";
```

这同巴尼在 shell 中使用 ln –s dodgson carroll 的结果是一样的。图 12-3 演示了其结果，包括索引节点 7033 中的诗歌。

现在如果巴尼选择阅读*/home/barney/poems/carroll*，这和从*/home/barney/pomes/dodgson* 中得到的结果一样的，因为系统会自动从符号连接中转过去。这个新名字不是文件的“真”名，因为（如你在图中所见）索引节点 7033 的计数仍为 1。这是因为符号连接告诉系统，“如果查找 *carroll*，现在应当离开，而去找 *dodgson*。”

符号连接可以自由的跨越不同的文件系统，可以给目录新的名字，而硬连接（hard link）做不到。符号连接可以指向任何的文件名，一个在这个目录，一个在另一个目录，甚至是不存在的文件。但这也意味着软连接不能像硬连接那样防止数据丢失，因为软连接和连接数无关。如果巴尼想删除 *dodgson*，系统则不能再从符号连接中得到相应的数据◆。虽然这里还有 *carroll*，从中读取数据将得到像 file not found 这样的错误信息。文件检测 –l "carroll"的结果为 true，但 –e "carroll"的结果为 false。它是个符号连接，但其并不存在。

图 12-3 索引节点 7033 的一个符号连接

inode 7033 1

Yet what are all such gaieties to me
Whose thoughts are full of indices
and surds?

$x^2 + 7x + 53$

=11/3

inode 919 2

.	919
..	8002
dodgson	7033
chicken	613
abacus	11320
egg	613
carroll->dodgson	

◆删除 *carroll* 仅仅是删除这个符号连接（sysmlink）。

由于符号连接可以指向一个不存在的文件，因此它也可以在创建文件时使用。巴尼的大多数文件存放在他的主目录中，*/home/barney*，但它也常访问另一个目录，其名字特别长，难于输入：*/usr/local/opt/system/httpd /root-dev/users/staging/barney/cgi-bin*。因此，他设置了一个符号连接*/home/banrey/my_stuff*，来指向前面的长名字，现在能很容易的访问它了。如果他创建了一个新的文件（在他的主目录下）叫做 *my_stuff/bowling*，这个文件的真实名字是*/usr/local/opt/system/httpd/root-dev/user staging/barney/cgi-bin/bowlin*。下周，系统管理员将巴尼的这些文件移到 */usr/local/opt/internal/httpd/www-dev/users/s tageing/barney/cgi-bin*，巴尼重新定位符号连接，他的所有程序仍能很容易的在新的地方找到文件。

通常，你的系统中的*/usr/bin/perl*，*/usr/local/bin/perl*，或者两个均是符号连接，连接到你的系统中 Perl 的真实目录。这使移

到新版本的 Perl 中变得很容易。假如你是系统管理员，安装了新的 Perl，同时希望老的 Perl 仍能使用，因为你不想带来任何不便。当准备好升级时，你改变这些符号连接，每一个使用#!/usr/bin/perl 开头的程序使用新版本的 Perl。此时，你可以替换以前指向旧版本的 Perl 的符号连接。(如同任何优秀的管理员一样，在升级前应当告诉你的用户在*/usr/bin/perl-7.2* 上测试他们的代码，告诉他们在这一个月的升级期间他们仍能使用老版本的 Perl,只需将第一行改作#!/usr/bin/perl-6.1，如果需要的话。)

可能让人惊奇的是，硬和软连接均是有用的。许多 non-Unix 操作系统一个都没有，这种缺失让人很强烈的感受到了。在某些 non-Unix 系统中，符号连接可能被当作“快捷方式（shortcut）”或“alias(别名)”来实现的。查看 Perlport 的用户手册了解最新的信息。

要找出符号连接指向的地方，使用 readlink 函数。它会告诉你符号连接指向的地方，如果参数不是符号连接其返回 undef：

```
my $where = readlink “carroll”;            #得到 “dodgson”
my $perl = readlink “/usr/local/bin/perl” #可能得到 Perl 放置的地方
```

可以使用 unlink 删除任意类型的连接。现在，你知道这个操作为什么得到这个名字了吧。unlink 删除目录中给定文件名的实体，符号连接减 1，也有可能释放掉相应的索引接点。

12．10 创建和删除目录

在一个目录下创建新目录是很容易的。使用 mkdir 函数：

```
mkdir “fred”, 0755 or warn “Cannot make fred directory: $!”;
```

true 意味着成功，$!会在失败时被置值。

第二个参数，0755 是什么意思呢？这是给新创建目录设置的初始权限◆（你总是可以在后面修改)。其值是用一个八进制值进行设置的，因为这个值会被当作 Unix 权限值进行解释，而它是按照一个组含有 3 个 bit 的值来进行的，而八进制数表示它非常恰当。是的，甚至在 Windows 或 MacPerl 上，使用 mkdir 函数也需要对 Unix 权限有所了解。权限 0755 是一个好的选择，因为它让你有所有的权限，而其他用户只有读的权限，不能进行任何修改。

◆通常权限值由 umask 值进行修改。参照 umask(2)了解更深入的信息。

mkdir 函数不需要你按照八进制值进行设定。他只是寻找一个数字（纯数字，或计算表达式)，除非你能立刻指出八进制的 0755 是十进制的 493，那最好让 Perl 来计算。如果不小心省略掉了开头的 0，得到十进制的 755，它对应的八进制是 1363，这是个古怪的权限值。

如我们在第二章所见，一个字符串的值被作为数字时是不会被当作八进制来解释的，无论其前面是否有 0。因此下面的不能正常工作：

```
my $name = “fred”;
my $permissions = “0755”;      #危险…不能工作
mkdir $name, $permissions;
```

噢，我们创建的目录的权限是古怪的 01363，因为 0755 被当作十进制来解释。要修正这种问题，可以使用 oct 函数，它会强迫将某个字符串按照八进制来解释，无论其前面是否有 0：

```
mkdir $name, oct($permissions);
```

如果要在程序中直接指定权限值，使用数字来代替字符串。需要 oct 函数的情况，通常是此值来源于用户的输入。假定我们从命令行中得到此参数：

```
my ($name, $perm) = @ARGV;     #前面两个参数是名字和权限
mkdir $name, oct($perm) or die “cannot create $name: $!”;
```

此处$perm 的值，最初会被当作字符串来解释，使用 oct 函数后会被作为八进制来解释。

要删除一个空的目录，按照类似于 unlink 函数的方法使用 rmdir 函数：

```
rmdir glob “fred/*”;                    #删除 fred/下面所有的空目录

foreach my $dir (qw(fred barney betty)){
  rmdir $dir or warn “cannot rmdir $dir: $!\n”;
}
```

和 unlink 一样，rmdir 返回删除的目录个数，如果一次删除一个，则会在失败时设置合理的$!值。

rmdir 在目录非空时失败。第一遍时，可以尝试使用 unlink 删除目录下的文件，其次再删除现在应当是空的目录。例如，假设我们需要一个地方，当程序运行时能写入大量的临时文件：

```
my $temp_dir = “/tmp/scratch_$$”;
mkdir $temp_dir, 0700 or die “cannot create $temp_dir: $!”;      #基于进程 ID

#将$temp_dir 作为所有的临时文件存放的地方

unlink glob “$temp_dir/* $temp_dir/.”;                    #将$temp_dir 里面的文件删掉
rmdir $temp_dir;                                          #将现在是空的目录删掉
```

初始化的临时目录名包括当前的进程 ID，这对每一个运行的进程均是唯一的，可以通过变量$$得到（类似于 shell）。我们这样做的目的是防止和其它进程冲突，只要它们也包括进程 ID 作为目录名的一部分就不会有问题。（使用程序名和进程 ID 是很普遍的，因此如果程序叫做 quarry，则目录通常被称为/tmp/quarry_$$。）

程序的末端，unlink 删除此临时目录下的所有文件，然后 rmdir 删除空目录。但是，如果我们在目录下创建了子目录，那 unlink 和 rmdir 均失败。要得到更可靠的解答，参看标准发布所附带的 File::Path 模块中的 rmdir 函数。

12．11 修改权限

Unix 的 chomd 命令可以改变文件或目录的权限。同样的，Perl 的 chomd 函数也能完成这种任务：

```
chmod 0755, “fred”, “barney”;
```

同许多的同操作系统接口的函数一样，chmod 返回其成功改变的个数，如果使用单个参数，$!会被设成恰当的失败原因。第一个参数是 Unix 权限值（甚至在 non-Unix 的 Perl 中）。基于我们在前面描述 mkdir 的原因，这个值通常是八进制的。

Unix 的 chomd 命令可以使用的符号权限（如+x 或 go=u-w）对 chomd 函数是无效的◆。

◆除非你从 CPAN 上下载并安装了 File::chmod 模块，它可以更新 chmod 操作使之可以理解这些符号值。

12．12 改变所有者

如果操作系统允许，你可以使用 chown 函数改变一批文件的所有者及所在的组。所有者及组是同时改变的，它们两个分别有一个数字值 user-ID 及 group-ID。例如：

```
my $user = 1004;
my $group = 100;
chown $user, $group, glob “*.o”;
```

如果你有一个像 melryn 这样的用户名，而没有数字，怎么办？调用 getpwnam 函数，将名字转换为数字，而对应的 getgrnam◆将组名转换为数字：

◆这两个名字基本上是最丑陋的。但不要因为这责备 Larry；因为 Berkley 的小子（发明 Unix 的人－译者注）是这样命名的。

```
defined(my $user = getpwnam “merlyn”) or die “bad user”;
defined(my $group = getprnam “users”) or die “bad group”;
chown $user, $group, glob “/home/Merlyn/*”;
```

defined 函数验证返回值是否为 undef，如果请求的 user 或 group 不存在，则返回 undef。

chown 函数返回其改变的文件的个数，而错误值被设置在$!之中。

12．13 改变时间戳

在极少情况下，希望欺骗别的程序，关于文件最近修改的时间，以及被访问的时间，你可以使用 utime 函数来做到这些。前面两个参数给出最近访问时间，和修改时间，剩下的参数是需要改变这些值的文件列表。时间按照内部的时间戳格式（和我们在第十一章中提到的 stat 函数返回值的类型是一致的。）

一个时间戳方便使用的值是"现在(right now)"，time 函数返回的值，其格式是合适的。更新当前目录下的所有文件，使它们看起来是昨天修改的，而访问时间为现在，可以如下做到：

```
my $now = time;
my $ago = $now -24*60*60;      #一天的秒数
utime $now, $ago, glob "*";       #设成当前访问的，一天之前修改的
```

没有什么限制你创建一个文件，其时间戳是在将来还是在过去（但受到 Unix 时间戳值的限制，必须在 1970 到 2038 年之间，或者你的 non-Unix 系统的限制，除非使用的是 64 位的时间戳）。也许可以使用它来创建一个目录，其中你做好注释，到时将写的小说传送出去。

第三个时间戳（ctime 的值）永远是 "now"，无论什么时候改变的文件，因此不能通过使用 utime 函数来设置它（当你设置它后，它会自动重置到 "now"）。因为其首要的目的是为了不断的备份：如果文件的 ctime 比备份磁带上的日期更新，则应当进行备份了。

12．14 练习

本节的程序可能是危险的。在一个空的目录里面测试它们，尽量避免意外的删除有用的信息。

答案参见附录 A：

1. [12]写一个程序要求用户输入一个目录名，再改变到那个目录去。如果用户输入的值是空白，则转变到他/她的主目录去。改变后，将这个目录下的普通内容（不包括有点（.）开头的项）按照字母顺序列出来。（提示：使用目录句柄还是 glob 更方便？）如果没有成功改变目录，提示用户，但不要尝试输出目录里的内容。

2. [4]修改程序，使之包含所有的文件，不仅仅是那些不以点(.)开头的文件。

3. [5]如果你在前面的练习中使用的是目录句柄，使用 glob 重写它。如果使用的是 glob，则用目录句柄重写。

4. [6]写一个类似于 rm 的程序，删除命令行中出现的任何文件。（不需要处理像 rm 中的选项）

5. [10]写一个类似于 mv 的程序，将命令行中的第一个参数重命名为第二个参数。（不需要处理像 mv 中的选项，或别的参数。）允许它是一个目录；如果是，使用相同的基本名字(basename)，生成新的目录。

6. [7]如果你的操作系统支持，写一个类似于 ln 的程序，能建立命令行中第一个参数的硬连接(hard link)到第二个参数。（不需要处理 ln 的选项，或者更多的参数。）如果系统没有硬连接，那么输出一条消息，指出如果可行你将进行的操作。提示：这个程序和前面的有类似的地方，注意到这一点能节约你编码的时间。

7. [7]如果你的操作系统支持，修改上一个练习的程序，使之支持 –s 选项（在别的参数前面），来表明你想创建一个软连接(soft link)而非(hard link)。(甚至在你没有硬连接的情况，你也有可能能创建软连接。)

8. [7]如果你的操作系统支持，写一个程序来查找当前目录下的符号连接（symbolic links），并将它们的值打印出来（如 ls –l 一样：name -> value）。

第十三章 字符串和排序

Perl 被设计成 90%擅长处理文本，10%处理其余的问题。因此 Perl 有强大的文本处理能力，包括正则表达式。但有时正则表达式过于复杂，你需要一种更简单的方法来处理字符串，这将在本章介绍。

13．1 使用索引寻找子串

查找的方法依赖于查找的地方。如果在一个大字符串中查找，那很幸运的，index 函数可以帮你的忙。其看起来如下：

```
$where = index($big, $small);
```

Perl 查找子串第一次在大字符串中出现的地方，返回第一个字符的位置。字符位置是从 0 开始编号的。如果子串在字符串的开头处找到，则 index 返回 0。如果一个字符后，则返回 1，依次类推。如果子串不存在，则返回-1◆。在本例中，$where 为 6。

◆对于早期的 C 程序员会注意到这类似于 C 语言的 index 函数。现在的 C 程序员可能知道，但从本书观点来看，你应当属于前者。

```
my $stuff = “Howdy world!”;
my $where = index($stuff, “wor”);
```

另一种方法是，可以将位置数（position number）想象成找到此子串所经过的字符。由于$where 为 6，则在$stuff 中找到 wor 前经过了 6 个字符。

index 函数总是报告子串出现的第一个位置。可以使用可选的第三个参数要求它从后面的某个地方开始查询，它会告诉 index 从什么位置开始：

```
my $stuff = “Howdy world!”;
my $where1 = index($stuff, “w”);              #$where1 得到 2
my $where2 = index($stuff, “w”, $where+1);    #$where 得到 6
my $where3 = index($stuff, “w”, $where+1);    #$where 为-1(没有找到)
```

(通常不能重复的查找某个子串，除非使用了循环。)第三个参数给出了返回值的极小值；如果自那个位置及之后的地方不能找到子串，则返回-1。

有时，你可能想知道某个子串最后出现的位置◆。可以使用 rindex 函数来做到。在下例中，我么查询最后一个斜线，它出现在字符串的位置 4：

◆它不是最后找到的地方。Perl 从字符串另一端开始查找，它返回第一次找到的位置，不过它们是相同的结果。其返回值和我们前面讨论的是一致的，也是从 0 开始编号。

```
my $last_slash = rindex("/etc/passwd", "/");    #值为 4
```

rindex 函数也有可选的第三个参数；此时，它给出的是可能的最大值：

```
my $fred = "Yabba dabba doo!";
my $where1 = rindex($fred, "abba");                    #$where1 得到 7
my $where2 = rindex($fred, "abba", $where - 1);    #$where2 得到 1
my $where3 =rindex($fred, "abba", $where2-1);      #$whrere3 得到-1
```

13．2 使用 substr 操作子串

substr 只处理部分的字符串。看起来如下：

```
$part = substr($string, $initial_position, $length);
```

它有三个参数：一个字符串，一个从 0 开始编号的初始位置（类似于 index 的返回值），以及子串的长度。返回值是一个子串：

```
my $mineral = substr("Fred J. Flintstone", 8, 5);       #得到"Flint"
my $rock = substr "Fred J. Flintstone", 13, 1000;      #得到"stone"
```

在上例中，如果请求的长度（例子中为，1000）超过了字符串的长度，Perl 不会有任何抱怨信息，但得到的是一个比你所希望的更短的结果。如果想明确要求到达字符串的结尾处，无论其或长或短，可以像下例那样省略掉第三个参数（参数）：

```
my $pebble = substr "Fred J. Flintsone", 13;    #得到 "stone"
```

初始位置可以是负的，表示从字符串结尾处开始（此时，-1 表示最后一个字符）◆。在下例中，位置-3 表示倒数第三个字符的位置，也就是字符 i 的位置：

◆这和在第三章中见到的数组索引是类似的。正如数组可以从 0 （第一个元素）开始，也可以从-1（最后一个元素）开始。子串的位置也可以从 0 开始（第一个元素）或者从-1（最后一个元素）开始。

```
my $out = substr ("some very long string", -3, 2);     # $out 得到 "in"
```

index 和 substr 可以很好的一起工作。在本例中，我们提取出了字符串中字母 l 后的子串。

```
my$ $long = "some very very long string";
my $right = substr($long, index($long, "l"));
```

现在我们要介绍一些非常酷的东西：字符串中选择的相应位置是可以改变的，如果字符串为变量◆：

◆是的，技术上而言，它可以是任何的左值(*lvalue*)。这个术语严格意义的介绍超出了本书的范围，但你可以将它想象成在标量赋值中任何

可以在赋值号(=)左边出现的。这通常是一个变量，但也可以（你马上将看到）是 substr 调用。

```
my $string = "Hello, world!";
substr($string, 0, 5) = "Goodbye";     # $string 现在变成了 "Goodbye, world!"
```

赋值的（子）串长度不需要和它替换的子串长度相同。字符串会自动调整到合适的长度。如果这还不能能给你留下印象，你还可以使用绑定操作符(=~)来将此运算限制在字符串的某一个部分。下例将字符串的最后 20 个字符串中的 fred 替换成 barney。

```
substr($string, -20) =~ s/fred/barney/g;
```

在我们自己的代码中，从没有用到过上述功能，因此你也很可能用不上。但知道 Perl 能完成比你需要完成的更多的工作，是一件很开心的事，不是吗？

substr 和 index 能完成的大部分工作都可由正则表达式来完成。在更适合使用它们的地方就使用它们。substr 和 index 通常更快，因为没有使用正则表达式：它们从不是大小写无关的，他们没有元字符(metacharacters)需要担心，也不设置内存变量(memory variables)。

除了给 substr 赋值外（第一次看起来有些怪异），也可以用更传统的方法◆来使用 susbtr：使用 4 个参数，第四个参数是替换的字符串：

◆按照惯例，我们的意思是“函数调用”的观点，而不是“Perl”的观点，因为这个功能在 Perl 的早期就有了。

```
my $previous_value = substr($string, 0, 5, "Goodbye");
```

我们也得到了相同的结果，当然这个函数也可在 void context 中使用，将结果丢弃。

13．3 使用 sprintf 格式化数据

sprintf 函数的参数和 printf 的参数完全相同（除了可选的文件句柄外），但它返回的是被请求的字符串，而非打印出来。这对于希望将某个格式的字符串存入变量以供将来使用的情况非常方便，或者你想比 printf 提供的方法，更能控制结果：

```
my $data_tag = sprintf
  "%4d/%02d/%02d %02d:%02d:%02d",
  $yr, $mo, $da, $h, $m, $s;
```

在上例中，$date_tag 得到像“2038/01/19 3:00:08”这样的值。格式字符串（sprintf 的第一个参数）在某些格式化数字前使用了前置 0，我们在第五章中讨论 printf 格式时没有提到。格式化数字中的前置 0 的含义是，如果需要，在前面加上 0，使之达到需要的宽度。格式中如果没有前置 0，则日期－时间的结果字符串中，我们得到不需要的空格，而非 0，看起来像“2038/ 1/19 3: 0: 8”。

13．3．1 在“货币数字”中使用 sprintf

一种常用 sprintf 的地方是，你需要给定数字的小数点后面几位，例如需要将某个金钱数量显示为 2.50 而非 2.5 也不是 2.49997！这使用“%.2f”很容易做到：

```
my $money = sprintf “%.2f”, 2.49997;
```

舍弃的部分是微不足道的，但在许多情况下，需要将所有有效的精度均保存在数字中，只是在输出时再进行四舍五入。如果“金钱数”太大，以至于需要逗号(,)分隔开，你会发现使用像下面这样的子程序将带来方便◆。

◆是的，我们知道并非全世界的任何地方都使用逗号来将数字分隔开。也并非任何地方均是 3 个数字一组，使用同美国一样的货币符号。但无论怎样，这里是一个好例子！

```
sub big_money {
   my $number = sprintf “%.2f”, shift @_;
   #在 do-nothing 循环中，每一次加入一个逗号
   1 while $number =~ s/^(-?\d+) (\d\d\d)/$1,$2/;
   #将美元符号放入合适的位置
   $number =~ s/^(-?)/$1\$/;
   $number;
}
```

这个子程序中使用了一些新技术，我们将在下面解释。子程序第一行将第一个（唯一的）参数进行格式化，使之小数点后有其只有两位。因此，如果参数中的数字为 12345678.9，则$number 为“12345678.90”。

下一行中使用了 while 语句。正如我们在第十章中所介绍的，上述代码可以按照传统的 while 循环重写：

```
while ($number =~ s/^(-?\d+) (\d\d\d)/$1, s2/) {
   1;
}
```

上述代码完成了什么工作呢？只要替换部分返回 true（表明成功），则循环就会持续下去。但循环部分什么也没做。这对于 Perl 来讲是正确的，但这告诉我们这段语句的目的是执行条件判断部分（替换操作），而非循环体本身。值 1 是这种用法中的一般方法，也可以使用任何别的值，它们是等价的◆。下面的例子和上面的结果一样：

◆换句话说，这是没有用的。顺便提一句，Perl 会优化这个常值表达式，使之不占用任何运行时间。

```
‘keep looping’ while $number =~ s/^(-?\d+) (\d\d\d)/$1,$2/;
```

现在知道了替换部分是上述循环的真正目的，但上述替换操作完成什么工作呢？$number 此刻是像“12345678.90”这样的数字。模式会匹配字符串的第一部分，但不能跳过小数点。(知道为什么不行吗？)变量$1 将得到“12345”,$2 得到“678”，现在$number 变成了“12345,678.90”(记住，它不能匹配小数点，因此最后一部分没有改动。)

知道模式开头处的破折号（负号：-。英语中破折号和负号相同。译者注）的含义吗？（提示：这个破折号只允许在字符串的一个地方出现。）我们将在本节结尾处指出，如果你还不清楚的话。

上述替换语句还没结束：由于替换成功，这个 do-nothing 又继续循环。最后，模式不能再匹配逗号前面的字符串了，$number 变成了“12,345,678.90”。替换部分每循环一次就加入一个逗号到数字中。

对于循环部分，它还没结束。由于上一次替换时成功的，我们将执行循环，然后再进行条件判断。但此时，模式不能匹配上了，因为它至少要匹配上 4 个数字，因此循环结束。

我们为什么不能使用/g 修饰符来进行一个“全面的”查找-替换，以防止使用容易混淆的 1 while 循环呢？我们不能用它，因为我们是从小数点开始（向前），而非从字符串开始。如此将逗号加入数字，是不能仅仅使用 s///g 替换的◆。你知道破折号的含义了吗？它允许字符串前面有负号。下一句代码也是这个原因，将美元符号加入恰当的位置，如“$12,345,678.90”或者在负数时“-$12,345,678.90”。美元符号可能不是字符串的第一个字符，否则这行代码要简单许多。最后，最后一行返回的格式化后的货币数字，可以将它输出在年度报表中。

◆至少，除非使用我们没有告诉你的一些高级的正则表达式技术，否则是不能完成它的。这些 Perl 的开发者，越来越使 Perl 书籍的作者难于使用“不能”这个字眼了。

13．4 高级排序

在第三章中，我们向你演示了如何使用内嵌的 sort 函数将列表按 ASCII 的升序排序。如果需要数字排序，或大小写无关的排序呢？也许你想将存放在 hash 中的元素排序。是的，Perl 允许你按你想要的方式将列表排序；在本章结束前你会看到所有这方面的例子。

你告诉 Perl 你要的顺序，通过排序定义（*sort-definition*）的子程序。当听到术语“sort subroutine(排序子程序)”，如果你上过任何计算机课程的话，可能冒泡排序，快速排序立刻呈现在你的脑海之中，然后说：“不，不要再来了！”。不要担心，不会那么坏。事实上，很简单。Perl 知道如何将元素排序，但它不知道你想要什么样的排序。因此排序子程序告诉它你要的顺序。

为什么要这么做呢？如果你想象一下，排序是将一些东西通过比较按顺序排列。由于一次不能将所有的元素进行比较，你需要一次比较两个元素，最终通过这两个元素的比较结果，来得到最终的结果。Perl 了解这一切，除了不知道你想怎样比较这两个元素，那就是需要你写的。

这意味着排序子程序不需要对大量的元素进行排序。事实上只需要比较两个元素。如果能将两个元素按恰当的顺序排序，Perl 则能够知道（重复的访问排序子程序）你想如何排序这些数据。

排序子程序被定义成普通的子程序一样（恩，是的，几乎一样）。这个子程序会被重复的调用，每一次检查一对你需要排序的列表元素。

现在，如果你写一个程序，期望得到需要排序的两个参数，你可能写出如下的：

```
sub any_sort_sub {      #这种方法不能工作
  my($a, $b) = @_;      #得到并对这两个参数命令
```

```
  #开始比较$a 和$b
  …
}
```

但排序子程序会被重复调用，通常是上百次或上千次。在顶端声明变量$a 和变量$b，再给它们赋值只需要很少的时间，但将这个时间乘以其被调用的次数，上百次，则它对程序的运行时间有显著的影响。

我们不希望这样。（事实上，如果这样做，并不能得到正确结果。）事实上，看起来在我们的子程序运行前，好像 Perl 为我们作了这项工作。你的排序子程序不需要第一行；$a 和$b 会自动被赋值。当排序子程序开始运行时，$a 和$b 会得到原始列表中的两个元素。

排序子程序返回一个值，表明这两个元素如何比较的（如 C 的 qsort(3)的，但为 Perl 自己的内部实现方式）。如果在最终结果中$a 出现在$b 之前，则其排序子程序返回-1。如果$b 出现在$a 之前，则返回 1。

如果$a 和$b 的顺序无关紧要，则子程序返回 0。为什么它无关紧要呢？也许你正在做一个大小写无关的排序，而这两个字符串是 fred 和 Fred。也许你正在做一个数字排序，而这两个元素相等。

我们可以写一个关于数字的排序子程序，如下：

```
sub by_number {
  if($a < $b){-1} elsif($a > $b){1} else {0}
}
```

要使用一个排序子程序，将它（不使用&）放在关键字 sort 和你要排序的列表之间。这个例子将一列数字按照数字顺序将其排序，并将结果放入@result 中：

```
my @result = sort by_number @some_numbers;
```

我们调用子程序 by_number，因为它描述了如何排序。但更重要的是，你可以这样阅读上述代码“sort by number(按照数字顺序排序)”，和在英语中一样。许多的排序子程序的名字都由 by_开头，来描述如何排序。基于同样的理由，我们可以将这个子程序叫做 numerically，但这需要输入更多的字符，且更可能输错一些字符。

我们不需要在排序子程序中申明$a 和$b，以及给它们设置。如果做了，子程序将不能得到正确结果。我们让 Perl 为我们给$a 和$b 赋值，我们只需要写如何比较。

事实上，我们可以使之更简单，效率也更高。因为这种三向的比较(three-way comparison)使用很频繁，Perl 提供了一种方便的简写方式。针对本例，我们使用太空船（spaceship）符号（<=>）◆。这个操作符比较两个数字，按照数字将其排序，并返回-1, 0, 1。因此，我们可以将排序子程序，写成下面更好的方式：

◆我们这样叫它的原因是，因为它很像星球大战（*Star Wars*）中的 Tie-fighters。是的，我们觉得它们很像。

```
sub by_number {$a <=> $b }
```

由于这个太空船符号(<=>)比较数字，你可能猜想有一个对应的针对字符串的三向操作符：cmp。这两个操作容易记忆和使

用。<=>操作符有类似的数字比较符，如>=，但它(<=>)是三个而非两个字符长，因为它有三个而非两个可能的返回值。而 cmp 操作符有些类似于字符串比较操作符如 ge，但它是三个而非两个字符长，因为它有三个而非两个可能的返回值◆。当然，cmp 的顺序和默认的排序顺序是一样的。你不需要书写这些子程序，因为它们几乎都是默认的排序顺序◆：

◆这不是巧合。Larry 作事总是带有目的的，让 Perl 易于学习和记忆。他是一个语言学家，因此他研究过人们如何思考语言。

◆你决不会写这些，除非你写一个初级的 Perl 教本，需要将它作为一个例子。

```
sub ASCIIbetically {$a cmp $b} my @string = sort ASCIIbetically @any_strings;
```

也可以使用 cmp 来创建更复杂的排序，如大小写无关的排序：

```
sub case_insenstive {"\L$a" cmp "\L$b" }
```

这里，我们比较两个字符串$a（强制转变成小写）和$b（强制转变成小写），给出一个大小写无关的排序。

我们没有修改这些数据本身，我们使用它们的值。这非常重要：出于效率的考虑，$a 和$b 不是这些数据的副本。它们是这些原始数据的新的，临时别名，因此如果修改了它们，则会破坏原始的数据。不要那样做，这几乎不被支持，也不被推荐。

当你的排序子程序像我们这里的那样简单时（大多数时候，是这样的），你可以通过使用"in line"子程序来代替排序的名字，使之更简单，如：

```
my @numbers = sort { $a <=> $b } @some_numbers;
```

在当代的 Perl 程序中，几乎见不到独立的排序子程序，你常见到的是像我们这里的那样：in line。

假设想按照数字的降序方式排序，这通过使用 reverse 能很容易的做到：

```
my @descending = reverse sort { $a <=> $b } @some_numbers;
```

这里有一个技巧。比较操作符（<=>和 cmp）是很"近视的"，它们不知道哪一个操作数是$a，哪一个是$b。它们只知道哪一个值在左边，哪一个在右边。如果$a 和$b 交换位置，比较操作符每一次得到的结果则是相反的。这意味着另一种得到逆序的方式：

```
my @descending = sort {$b <=> $a } @some_nubmers;
```

通过少许的练习，你可以容易的阅读这些代码。它是一个降序比较（因为$b 在$a 之前，这表示降序），它是数字比较（因为它使用的是<=>而非 cmp）。因此，这是一个按相反方向的数字排序。在当代的 Perl 版本中，使用哪一种方法无关紧要，因为 reverse 被当作 sort 的一个修饰符，特殊的简写方式阻止你使用一种方式排序但得到的却是另一种结果。

13．4．1 依据值对 Hash 进行排序

一旦对列表进行了排序，你就进入了一种境地：想根据 value 对 hash 排序。例如，我们中的三人昨晚去打保龄球，在下列的 Hash 中有他们的保龄球成绩。你想按照合适的顺序将它们打印出来，游戏的胜利者在顶端，因此我们想根据成绩（score）对 hash 进行排序：

```
my %score = (“barney” =>, “fred” =>205, “dino” => 30);
my @winners = sort by_score keys %score;
```

实际上并不能根据 score 对 hash 进行排序；这只是文字上的简写。你不能对 hash 排序。我们在前面的 hashes 中使用 sort 时，只是对 hash 的 keys 排序（按照字母表顺序(ASCIIbetical)排序）。现在我们将对 hash 的 keys 排序，其顺序由其对应的 hash 中的值决定。此时，结果是根据保龄球的成绩这三人的名字的有序列表，。

写出这个排序子程序是非常容易的。我们需要的是针对 score 而不是名字，使用数字比较。不是比较$a 和$b(选手的名字)，我们想比较$score($a)和$score($b)（它们的成绩）。如果你这样思考，答案则呼之欲出，如下：

```
sub by_socre { $score{$b} <=> $score{$a}}
```

让我们仔细的分析它，看看它是如何工作的。想象第一次调用它时，Perl 给$a 赋值 barney，给$b 赋值 fred。比较是$score{“fred”} <=> $score{“barney”}，它是（通过 hash 得到）205 <=>195。<=>是“近视的”，因此它发现 205 在 195 之前，则说：“不，这不是正确的顺序；$b 应当在$a 之前。这告诉 Perl fred 应当在 barney 之前。

可能下一次调用子程序时，$a 是 barney，而$b 是 dino。“近视的”数字比较符看到的是 30<=>195，因此报告它们是正确的顺序：$a 确实是在$b 之前。因此，barney 在 dino 之前。此刻，Perl 有了足够的信息来得到列表的顺序：fred 是胜利者，barney 第二名，而 dino 第三。

为什么比较运算中$score{$b}在$score{$a}之前，而非别的方式？因为我们想按保龄球成绩的降序排列，从最高成绩依次向下。你可以（经过一些训练）一眼就能读懂这些代码：$score{$b} <=> $score{$a}的意思是根据 score，将它们按照数字逆序排序。

13．4．2 对多个 keys 排序

我们忘了昨晚第四个选手也和其他三个参加了保龄球比赛，此时 hash 看起来如下：

```
my %score = {
 “barney” =>95, “fred” => 205,
 “dino” =>30, “bam-bamm” => 195;
};
```

bam-bamm 和 barney 的成绩相同。哪一个选手应当在排序后的列表的前面呢？比较运算不能给出结果（发现两边的成绩相同）因为它比较这两个值得结果是 0。

也许这无关紧要，但我们需要有一个严格定义的顺序。如果几个选手有相同的成绩，我们希望他们在结果的列表中出现在一块，但在这一块中，他们是按字母排序的。怎样书写我们刚才所描述的排序子程序呢？同以往一样，结果很简单：

```
my @winners = sort by_score_and_name keys %score;
sub by_score_and_name {
  $score{$b} <=> $score{$a}        #按照降序的成绩
      or
  $a cmp $b;                       #字母顺序的名字
}
```

这是如何工作的呢？如果<=>发现了两个不同的成绩，这个比较就是我们所需要的。它返回-1 或 1，一个 true 值，因此低优先级的 or 意味着后面的表达式将被忽略，我们需要的比较被返回。（记住 or 返回最后一个被估值的表达式。）如果<=>发现两边的结果一致，则得到 0，一个 false 值，cmp 操作派上了用场，根据这些 key(字符串)返回恰当顺序的值。因此，如果成绩相同，则根据字母顺序排列。

我们知道当使用像这样使用排序子程序时 by_score_and_name，其永远不可能返回 0。（你知道为什么吗？答案在脚注中◆。）我们知道排序的顺序总是被良好的定义的；也即是说，同样的数据，今天排序的结果和明天排序的结果是一样的。

◆返回 0 的唯一的情况是，这两个字符串相同，但（由于这些字符串是 hash 的 keys）我们知道它们是不同的。如果你将有重复（相等的）字符串的列表传给 sort，其结果可能为 0。但这里我们传入的是 hash 的 keys。

排序子程序并没有限制在只有两级。这里有一个 Bedrock 图书馆程序，根据 5 级的排序将赞助者的 ID 号排序◆。这个排序例子根据每一个赞助者的 outstanding fines（由一个叫做&fines 的子程序计算，这里没有)，他们当前借出的数目（%items），他们的名字（首先根据姓（family name），再根据名(personal name)，均来自于 hash），最后根据赞助者的 ID 号：

◆在当代的 Perl 程序中，使用 5 级的排序并非少见，但在以前极少使用。

```
@patrons_IDs = sort {
   &fines($b) <=> &fines($a) or
   $items{$b} <=> $items{$a} or
   $family_name{$a} cmp $family_name{$a} or
   $personal_name{$a} cmp $family_name{$b} or
   $a <=> $b
} @patron_IDs;
```

13．5 练习

答案参见附录 A：

1. [10]写一个程序，读入一串数字，将它们按照数字排序，将结果按右对齐的列打印出来。使用下面的数据进行检测：

```
17   000   04   1.50   3.14159   -10   1.5   4   2001   90210   666
```

2. [15]写一个程序，将下例 hash 数据根据姓（last name）按照大小写无关的字母顺序进行排序，并把结果打印出来。当 last name 相同时，再按照名(first name)排序（不用关心大小写）。因此，第一个输出的的名字是 Fred's，最后一个是 Betty's。具有相同 family name 名字在一起。不要改变数据。输出名字的大小写应当和这里的一样。

```
my %last_name = qw{
    fred flintstone Wilma Flintstone Barney Rubble
    betty rubble Bamm-Bamm Rubble PEBBLES FLINTSONE
};
```

3. [15]写一个程序，查找给定子串在给定字符串中出现的每一个位置，输出子串出现的位置。例如，给定字符串为"This is a test." 给定子串为"is",它应当输出 2 和 5。如果子串是"a"，它应当输出 8。如果子串为"t"呢？

第十四章 进程管理

作为程序员的一个好处是，可以调用别人的代码，而不用自己编写。是时候学习如何管理你的孩子了◆，由 Perl 直接调用其它程序。

◆也就是，子程序。

同 Perl 中其它地方一样，有不止一种处理方法（There's More That One Way To Do It），存在许多交叉，变种等等。如果不喜欢第一种方法，那么继续阅读，找到一种你喜欢的解决方法。

Perl 是非常容易移植的。本书大多数地方都不需要注释说明这种方法在 Unix 系统上运行，另一种在 Windows 上，第三种在 VMS 上。但是，如果你调用其它的程序，那么 Macintosh 上面的程序则可能和 Cray 上是不同的。本章的例子基本上是基于 Unix 的；如果你的系统是 non-Unix 系统，有可能存在差别。

14．1 系统函数

在 Perl 中调用子进程来运行程序的最简单方法是使用 system 函数。例如，要调用 Unix 的 date 命令，看起来如下：

```
system “date”;
```

子进程运行 date 命令，它继承了 Perl 的标准输入，标准输出，标准错误。这段话意思是 date 命令产生的日期时间字符串被放在 Perl 的 STDOUT 输出的地方。

系统函数的参数和通常在 shell 中输入的是一样的。如果是复杂一些的命令，如 “ls –l $HOME”，我们也只需将它们放入参数中：

```
system ‘ls –l $HOME’;
```

我们这里将双引号变成了单引号，因为$HOME 是一个 shell 变量。否则，shell 看不到美元符号，因为 Perl 会将它用值进行替换。当然，我们也可以这样写：

```
system “ls –l \$HOME”;
```

但这样看起来非常笨拙。

date 命令只是输出的，假设是个有些饶舌的命令，首先询问你“你的时区是多少？”◆这会输出在标准终端上，程序再监听标准输入（从 Perl 的 STDIN 继承），等待回应。你看见这个问题，输入答案（例如“津巴布韦时间”），date 再完成剩下的操作。

◆我们所知道的范围内，没有一种 date 命令的实现是这样的。

当子进程运行时，Perl 会耐心地等待其结束。如果 date 命令花去 37 秒，则 Perl 会暂停 37 秒。可以使用 shell 提供的工具将它变成后台运行的进程◆：

◆我们这里所介绍的依赖于你的系统。Unix shell（/bin/sh）允许你在这类命令前加上&符号，使之变成后台进程。如果你的 non-Unix 系统不支持这种方法，那就不能这样做。

```
system “long_running_command with parameters &”;
```

这里是 shell 调用的，注意命令行结尾处的&，它将 long_running_command 变成后台运行的。然后 shell 立刻退出，Perl 注意到这些再继续执行。这种情况，long_running_command 是 Perl 进程的孙子进程，Perl 没有对它直接的了解。

当命令“非常简单”时，不会使用 shell。例如 date 和 ls 命令，请求命令会直接由 Perl 来调用，如果需要，它会查询 PATH ◆来查找命令。但如果其中有些怪异的地方（例如 shell 的元字符，如美元符号($)，分号(;)，竖线(|)），Perl 会调用标准的 Bourne Shell(*/bin/sh*◆)来处理这些复杂的内容。此时，shell 是子进程，被请求的命令是孙进程（或者更远的后代）。例如，你可以写一个如下的一个 shell 脚本：

◆PATH 在任何时候都可以被$ENV{‘PATH’}改变。初始时，这是从父进程（通常是 shell）继承的环境变量(environment varible)。改变这个值会影响到新的子进程但不能影响到之前的父进程。PATH 是一些可执行程序(命令)的目录，甚至是在 non-Unix 系统之中。

◆或者 Perl 被 build 时所监测到的。基本上，在类 Unix 系统中这是*/bin/sh*。

```
system ‘for i in *; do echo = = $i = =; cat $i; done’;
```

这里，我们也是使用的单引号，因为需要美元符号对 shell 有含义，而对 Perl 没有。双引号允许 Perl 用当前的值来替换$i 而不是让 shell 用它自己的值来代替。顺便说一句，这个小的 shell 脚本会遍历当前目录下所有的普通文件，输出每一个文件的名字和内容；如果不相信，你可以尝试一下。

14．1．1 避免 Shell

系统调用可能不止一个参数◆，如果这样，shell 将不会被调用，无论其有多么复杂：

◆或者有一个参数是 indirect-object slot，例如 system {‘fred’} ‘bamey’。它运行程序 barney,但实际上欺骗了它而是认为调用的是 ‘fred’。查看 perlfunc 的用户手册

```
my $tarfile = “something*wicked.tar”;
my @dirs =qw(fred/flintstone <barney&rubble> betty );
system “tar”, “cvf”, $tarfile, @dirs;
```

上例中，第一个参数（这里是“tar”）给出了一个通常能在 PATH 查询中找到的命令，余下的参数再一个接一个的进行。甚至如果参数中有对于 shell 有重要意义的字符，如$tarfile 中的名字，@dirs 中的目录名，shell 也不会有机会来处理它。因此，

tar 命令将得到这 5 个参数，和下面的比较：

```
system "tar cvf $tarfile @dirs"; #Oops!
```

这里我们将一些内容倾入 flintstone 命令，并将其作为后台运行，再打开 betty 作为输出。

这看起来让人惊慌，特别在这些变量是由用户输入的，例如从 web 输入，或别的情况。因此，如果可以将它按照多个参数的 system 版本一样，那你很可能应当使用这种方法来调用你的子进程。(你将不得不失去 shell 为你提供得一些功能，如设置 I/O 重定向，后台进程，以及类似的。在自由调用时，没有这些。)

注意，system 的单个参数的调用，几乎等价于多个参数的调用：

```
system $command_line;
system "/bin/sh", "-c", $command_line;
```

但没有人使用后一种版本，除非想在不同的 shell 中处理，如 C-shell：

```
system "/bin/csh", "-fc", $command_line;
```

甚至上面的也不常见，因为 One True Shell◆看起来更方便，特别是对于脚本而言。

◆也就是*/bin/sh*。或者你的 Unix 系统上最类似于 Bourne shell 的 shell。如果没有 One True Shell，Perl 会考虑调用别的命令行解释器，以得到最恰当的结果。查看 Perl 移植的文档。

系统调用的返回值基于子命令(child command)退出的状态◆。在 Unix 系统中，退出值 0 指一切正常，非 0 退出值通常表明什么地方出了错误：

◆它是 "wait"状态，为子程序的退出码(child exit code)乘以 256，如果内核出了问题需要加上 128，加上信号码(singal number)如果是它激发了中断。但我们几乎不会检查这样的细节，一个 true/flase 值对几乎所有的应用已经足够了。

```
unless (system "date") {
  #返回 0 —表示成功
  print "We gave a date, OK!\n";
}
```

这追溯于通常大多数操作所采用的"true is good, false is bad"策略，因此要按照典型的"do this or die"风格来写，我们要返回 false 或者 true。最简单的方法是在 system 操作前加上一个前缀的感叹号（逻辑非）：

```
!system "rm –rf files_to_delete" or die "something went wrong";
```

这个例子中，在错误信息中包含$!是不恰当的，因为失败很大可能是由于 rm 操作部分的原因，这不是和系统调用相关的错误，因此 Perl 的$!变量不能描述。

14．2exec 函数

除了一个地方（非常重要的）以外，我们前面介绍的 system 的语法（syntax）及语义(semantics)对于 exec 函数也是相同的。system 函数创建子进程，它会立刻去执行请求的操作，Perl 则暂停。exec 函数引起 Perl 自己处理请求的操作。可以将它看作“goto”而非子程序调用。

例如，假设我们想在*/tmp* 目录下运行 bedrock 命令，将-o args1 以及其它的调用的参数传递给它。看起来如下：

```
chdir “/tmp” or die “Cannot chdir /tmp: $!”;
exec “bedrock”, “-o”, @ARGV;
```

当执行到 exec 操作时，Perl 找到 bedrock，再“跳到那里(jumps to it)。”此时，Perl 进程离开◆，只有运行 bedrock 命令的进程。当 bedrock 结束时，Perl 不会回来，因此我们将得到退出的提示(prompt back)，如果在命令好中调用这个程序。

◆实际上，这是同一个进程，执行 Unix 的 exec(2)系统调用（或者等价的）。进程 ID 是同一个。

为什么这是有用的呢？如果 Perl 程序的目的只是设置一个特殊的环境变量，以运行另一个程序，则当另一个程序启动时，其目的就已达到。如果我们使用 system 而非 exec，我们就让一个 Perl 程序等待另一个程序的完成，当然 Perl 最终能够退出。但这会浪费资源。

说了这么多，实际使用 exec 的情况很少，除了和 fork 一起使用外（这将在后面介绍）。如果分不清 system 和 exec，那使用 system，在大多数情况下，都不会有问题。

由于当被请求的命令运行时，Perl 不再能控制，因此在 exec 命令后面的任何代码均不会有任何意义，除了处理不能启动该请求的错误的代码外：

```
exec “date”;
   die “date couldn’t run: $!”;
```

如果将 warning 打开，并且 exec 后面除了 die 还有别的代码◆，你将得到提示信息。

◆或者 exit。或者在块的结尾处。这可能在新的 Perl 版本中改变。

14．3 环境变量

当启动另一个进程时（利用这里讨论的任何技术），你可能需要将它们的环境变量进行不同的设置。你可以启动一个进程，其有某个工作目录(working directory,)，这从你的进程中继承。另一个配置的细节是环境变量。

其中最著名的环境变量是 PATH。（如果你从没有听过，那可能你的系统中没有环境变量。）在 Unix 或类似的系统中，PATH 是一个由逗号分隔开的目录列表，这些列表的元素可能是程序名。当输入像 rm fred 这样的命令时，系统将按照顺序在这些目录列表中查找 rm 命令。Perl(或你的系统)将在任何需要查找程序的时候使用 PATH。如果程序运行其它的程序，也需要在 PATH 中学找。（如果你是用完全的名字作为命令，如*/bin/echo*，则没有必要查找 PATH。但这没有那么方便。）

在 Perl 中，环境变量可以通过 %ENV 这个 hash 变量得到，hash 中的每一个 key 代表一个环境变量。当你的程序开始执行时，%ENV 中的值从其父进程中继承（这通常是 shell）。修改这个 hash 将改变环境变量，这将被新的进程继承，也有可能被 Perl 继承。例如，假定你想运行系统的 make 工具（它通常会运行别的程序），你想首先在某个私有目录中查找命令（包括 make）。并且让我们假定你在运行命令时不想设置 IFS 这个环境变量，因为它可能使 make 或别的子命令（subcommand）做一些错误的事。如下：

```
$ENV{'PATH'} = "/home/rootbeer/bin:$ENV('PATH')";
delete $ENV{'IFS'};
my $make_result = system "make";
```

新创建的进程通常会从它们的父进程中继承环境变量，当前的工作目录，标准输入，输出，以及错误流，还有一些深奥的东西。查看你系统上编程的书籍，以了解更详细的信息。(在大多数系统中，不能改变 shell 或者启动它的父进程的的环境变量。)

14. 4 使用反引号捕捉输出

对于 system 和 exec，其输出的结果传到 Perl 标准输出的地方。有时，将其结果作为字符串保留下来以便进一步处理是很有趣的。这可以通过使用反引号(`)而非单引号或者双引号做到：

```
my $now = `date`;                #捕获 date 的输出
print "The time is now $now";    #已经有换行符
```

一般，date 命令会输出大概 30 个字符到标准输出设备，给出当前日期，时间以及换行符。当我们把 date 放在反引号之间，Perl 执行 date 命令，将其作为一串字符串值，这里赋给变量$new。

这同 Unix shell 的反引号含义是类似的。但，shell 做了额外的工作，将结尾的换行符去掉了，方便在其它地方应用。Perl 很老实；它给出了实际的输出结果。因此在 Perl 中要得到相同的结果，需要使用 chomp：

```
chomp(my $no_newline_now = `date`);
print "A moment ago, it was $no_newline_now, I think.\n";
```

反引号中的值类似于单引号的 system◆的效果，其被作为双引号字符串解释，也就是其中的转义字符，变量会被处理◆。例如，要得到一列 Perl 函数的文档，可以重复使用 perldoc 命令，每一次使用不同的参数：

◆也就是说，它总是被 One True Shell(/bin/sh)或类似的处理，如果 system 调用一样。

◆如果想要反斜线，需要使用 2 个。如果想要两个（这在 Windows 系统中经常出现），则需要四个。

```
my @functions = qw { int rand sleep length hex eof not exit sqrt umask };
my %about;

foreach(@functions) {
```

```
    $about{$_} = `perldoc –t –f $_`;
}
```

当每一次调用时，$_的值都不同，使我们一次处理一个命令。如果没见过其中的一些函数，查找相应文档了解其功能是有益的。

反引号没有简单的单引号等价形式◆；变量，和转义字符（如：\n）是会被处理的。没有简单的多参数的 system 的等价形式，如果不使用 shell 的话。如果反引号中的命令过于复杂，Unix 的 Bourne Shell（或者你的系统使用的）被自动调用，解释命令。

◆有一些更复杂的方法，如可以将字符串放在 qx'...'之中等。

我们建议你在像本例那样部捕捉输出◆的情况下避免使用反引号：

◆这被叫做"void" context。

```
print "Starting the frobnitzigator: \n";
`frobnitz -enable`;                    #请不要这要做!
print "Done!\n";
```

问题在于 Perl 会尽力去捕捉这个命令的输出，虽然你是想将其丢弃。并且，你也失去了使用 system 对多个参数控制的机会。因此从安全和效率的观点，应当使用 system。

反引号命令的标准错误继承了 Perl 当前的标准错误输出。如果命令将错误信息传到了标准错误(standard error)，你可能在终端见到它，这可能让用户迷惑，因为他没有调用 frobnitz 命令。如果想用使用标准输出（standard output）捕捉错误信息，可以使用 shell 的"将标准错误并入当前的标准输出中"，其如通常在 Unix shell 中所作那样 2>&1：

```
my $output_with_errors = `frobnitz –enable 2>&1`;
```

这会将标准错误输出并入标准输出中，和在终端中所见的基本类似，只是可能由于缓冲，其顺序不同。如果想输出和错误输出分离，你会发现一种难于输入(hard-to-type)的解决方法◆。类似的，标准输入也继承了 Perl 的当前标准输入。在反引号中的命令通常不会从标准输入读，因此这几乎不是问题。但，让我们以 date 命令需输入时区（我们早些也作了这些假设）为例。这会引起问题，因为提示语"which time zone"将被送到标准输出上，而这会被捕获作为部分的值。然后，date 命令相从标准输入读入。由于用户没有见到提示符，他不知道需要输入。很快，用户就会通知你，告诉你你的程序不动了。

◆例如 Perl 标准的库 IPC::Open3。或者你自己写 forking，这会在后面介绍。

因此，不要使用那些需要从标准输入读入的命令。如果不太确定其是否从标准输入读入，加入从*/dev/null* 的重定向，如：

```
my $result = `some_questionable_command arg arg argh </dev/null`;
```

child shell 的输入会重定向到/dev/null，grandchild 的有询问的命令（questionable command）在最坏的情况下将从/dev/null 读入，当然什么也没有。

14．4．1 在 List context 中使用反引号

如果命令的输出有多行，在标量环境中使用反引号将其作为单个字符串返回，其中包括换行字符。但是，在列表 context 中使用相同的反引号字符串将得到一个列表值，每一元素含有一行值。

例如，Unix 的 who 命令通常会返回当前登录系统的每一个用户（一个用户一行），如下：

```
merlyn     tty/42     Dec 7    19.41
rootbeer   console    Dec 2    14.15
rootbeer   tty/12     Dec 6    23:00
```

左侧一列是用户名(username)，中间一列是 tty 名字（用户到机器的连接的名字），剩下的是登陆日期和时间（可能是远程登录的信息，但本例不是）。在标量 context 中，我们一次得到所有值，我们需要将其分开：

```
my $who_text = `who`;
```

但在列表 context 中，我们会自动得到按行分开的数据：

```
my @who_lines =`who`;
```

我们在@who_lines 会得到几个不同的元素；每一个由换行符结束。在其上使用 chomp 会去掉换行符。让我们用不同的方法来处理。如果我们将其传递给 foreach，会自动在行之间进行替代，依次传递给$_：

```
foreach (‘who’) {
   my($user, $tty, $date) = /(\S+) \s+(\S+)\s+(.*)/;
   $ttys{$user}.= “$tty at $date\n”;
}
```

它会循环 3 次。(你的系统可能在任意时刻有不止 3 个登录用户。) 循环中的第一个语句是正则表达式匹配，没有使用邦定操作符（=~），它会匹配$_，这是可以的，因为数据就存放在其中。

正则表达式寻找非空白词（nonblank word)，一些空白，非空白词，一些空白，以及此行剩余的部分，但不包括，换行符（因为点(.)默认的情况下不会匹配换行符）◆。这是可行的，因为$_每一次的值的结构是类似的。其中第一次循环时，匹配成功，$1 为“merlyn”，$2 为“tty/42”，$3 为“Dec 7 19:41”。

◆现在你知道为什么点(.)在默认时不匹配换行符了。它使我们容易的写出这样的模式，而不用担心结尾处的换行符。

但，这个正则表达式是在 list context 中使用的，因此我们会得到其列表值，而不是 true/false 值，表示其是否匹配，如在第八章中介绍的。因此，$user 的值为 “merlyn”，等等。

循环中的第二条语句将 tty 和 date 的信息存储到 hash 中，其值是附加(可能是 undef)，因为一个用户如上例的"rootbeer"可能登陆不止一次。

14．5 像文件句柄那样处理

到目前为止，我们只是介绍了处理同步进程（synchronous processes）的方法，此时仍然是 Perl 在控制，调入命令，（通常）等待其结束，可能捕获其输出。但 Perl 也可以在调用子进程时仍保持运行，与之交互◆，直到完成任务为止。

◆通过管道或者任何你的操作系统提供的进程间通讯的方法。

开始一个并发（并行）子进程的语法是将命令作为"文件名"传给 open，并在其前面或后面加上竖线（|），竖线是"管道（pipe）"符。基于这些原因，这通常叫做管道打开（*piped open*）：

```
open DATE, "date|" or die "cannot pipe from date: $!";
open MAIL, "|mail merlyn" or die "cannot pipe to mail: $!";
```

在第一个例子中，竖线在右边，命令被调入，且其被打开到 DATE 这个文件句柄进行读入，这和 shell 中的命令 date|your_program 类似。在第二个例子中，出现在左边，命令的标准输入被连接到 MAIL 这个文件句柄，这和命令 your_program | mail merlyn 类似。在任意种情况下，命令被调入，和 Perl 进程是独立的◆。如果子进程没有被创建成功，在打开时将失败。如果命令不存在或异常退出，在打开时将（通常）不会被当作错误，但在关闭时会。我们将会很快介绍到。

◆如果一个 Perl 进程在命令完成之前退出了，一个读入的命令将得到 end-of-file，当一个命令正在写入时在默认情况下下一次写入将得到"broken pipe"的错误信号(error signal)

为了各种目的和意图，程序剩余部分不知道，也不关心，这个句柄打开的是进程还是文件。因此，要从文件句柄中读入数据，我们可以使用通常的方法：

```
my $now = <DATE>;
```

要将数据送给 mail 进程（从标准输入等待要传递给 *merlyn* 的消息），一个简单的打印文件句柄就可以了：

```
print MAIL "The time is now $now"; #假定$now 由换行符结尾
```

简言之，你可以假装这些文件句柄指向一些特殊的文件，一个含有 date 命令的输出，一个将自动的使用 mail 命令将其邮寄出去。

如果一个进程连接到一个文件句柄，被打开来读，然后退出，文件句柄返回 end-of-file，如同读到普通文件的结尾符。当关闭一个打开来写入到进程的文件句柄时，进程将发现 end-of-file，因此，结束发送 email，关闭句柄：

```
close MAIL;
die "mail: nonzero exit of $?" if $?;
```

如果关闭一个关于进程的文件句柄，Perl 等待其完成，以便得到其进程的退出状态。退出状态可以通过变量$?得到（同 Bourne Shell 的变量一样）其数值同 system 函数的返回值是一致的：0 表示成功，非 0 表示失败。每一个新的退出的进程会覆盖以前的值，因此尽快保存它，如果需要的话。($?变量也含有最近的 system 或反引号(``)调用的退出状态，如果你好奇的话。)

这些进程是同步的，同管道命令一样。如果尝试读时，数据还没准备好，则进程被悬挂（不会消耗额外的 CPU 时间）直到数据准备好为止。同样的，如果一个写进程在读进程之前，写进程将慢下来，直到读进程跟上来。这里有一个缓冲（buffer）（通常是 8KB），因此它们不需要完全一致。

为什么将进程作为文件句柄来使用呢？这是将结果写入进程的唯一简单方法。如果只是读，反引号调用可以方便的处理，除非需要将其结果作为输入传给别的进程。

例如，Unix 的 find 命令根据文件的属性来定位文件，如果在大量的文件中查找，它会花一些时间。（例如，从根目录开始查找）。你可以将 find 命令放在反引号之间，当找到就显示将更好：

```
    open F, “find / -atime +90 –size +1000 –print|” or die “fork: $!”;
    while(<F>) {
        chomp;
        printf %s size %dK last accessed on %s\n”,
        $_, (1023 + -s $_)/1024, -A $_;
}
```

这个例子中的 find 命令查找所有在最近 90 天没有访问，且大于 1000 块的文件。（它们是被备份的好对象）当 find 查询时，Perl 等待。每当找到时，Perl 会将其名字，以及一些其它信息显示出来以供进一步的分析。如果用反引号来做，我们在 find 命令执行完之前是得不到任何输出的。通常在没有结束前就看见它在工作是很舒服的。

14．6 使用 fork

除了介绍的 high-level 接口外，Perl 还提供了直接访问 low-level 进程管理系统的方法。如果以前没有做过◆，你很可能想跳过此节。事实上将本节写成一章更恰当，但让我们快速的介绍这些：

◆或者你的系统不支持 forking。但 Perl 的开发者努力工作，甚至将 forking 移植到那些底层的进程模型和 Unix 很不同的系统之中。

```
    system “date”;
```

看看如何使用 low-level 系统调用的方式来完成它：

```
    defined(my $pid= fork) or die “Cannot fork: $!”;
    unless ($pid){
     #子进程在这里
     exec “date”;
     die “cannot exec date: $!”;
    }
    #父进程在这里
```

```
waitpid($pid, 0);
```

这里，我们检查 fork 的返回值，失败时为 undef。通常会成功，因此在下一行时有两个独立的进程，但只有父进程的$pid 非 0，因此只有子进程在执行 exec 函数。父进程跳过这部分执行 waitpid 函数，等待那个子进程结束（如果期间其它的进程完成，将被忽略）。如果这些看起来像官样文章，只需记住你仍可以继续使用 system 函数，而不用担心会被你的朋友笑话。

如果不怕麻烦，还能完全控制任意的管道，重新排列文件句柄(rearranging filehandles)，通知你的进程 ID 和你的父进程的 ID（如果可知的话）。但是，这些内容足够写一章了，请参阅 perlipc 的用户手册了解更详细的信息（以及关于你系统的应用程序编程的优秀书籍）。

14．7 发送和接收信号

Unix 信号是传送给进程的小消息。它不能说明太多问题；就像汽车的喇叭声：你听得喇叭声是指“注意，桥塌了”，“灯的颜色变了，走”，“停车，有个小孩在车顶上”，或“喂，你好”？幸运的是，Unix 信号比它容易理解，因为有不同的信号针对这些不同的情形◆。信号由名字（如 SIGINT, 意思是“interrupt singal(中断信号)”）和对应的一个小整数（范围 1 到 16，1 到 32，1 到 64，这要看你的 Unix 的情况）组成。信号通常是在某个事件发生时发送的，如在终端按下中断键（通常是 Ctrl-C），这会发送 SIGINT 给这个上的所有进程◆。某些信号会自动由系统发送，他们是由另一个进程发送的。

◆是的，实际上不是这样的情形，这里只是一个比喻。对于这些，信号是 SIGHUP, SIGCONT,SIGINT,和 SIGZERO(信号 0).

◆你认为按下 Ctrl-C 停止程序。事实上，它只是发送 SIGINT 信号，默认情况这会停止程序。在本章后面，你可以写个程序，当接收到 SIGINT 时它能做些不同的事，而非立刻停止程序。

你可以从你的 Perl 进程发送信号到其它的进程，但你需要知道目标进程的 ID。怎么得到它有些复杂◆，我们假设你已经知道了其 ID 为 4021，现在想发送 SIGINT 给它。这非常简单：

◆通常，你知道其进程 ID，因为它是你通过 fork 创建的子进程，或者从文件或者别的程序知道。使用别的程序可能困难和易犯错误，这就是为什么许多运行时间长久的程序将它们自己的进程 ID 保存在文件中，通常会在程序的文档中说明这些。

```
kill 2, 4201 or die “Cannot signal 4201 with SIGINT: $!”;
```

其名字是“kill”因为信号的一个主要目的就是停止长时间运行的程序。你可以使用字符串‘INT’替换 2，因为信号码（singal number）2 是指 SIGINT。如果进程不存在◆，你会得到一个错误的返回值，可以使用这种技术验证一个进程是否活着。特殊的信号码 0 的含义是：“检查是否可以发送信号，但我实际不想，所以什么也不发送。”这种进程探测的程序看起来如下：

◆如果不是超级用户，进程又是其他人的，你发送信号也会失败。毕竟，将 SIGINT 发送给其它用户的程序是非常不礼貌的。

```
while (kill 0, $pid) {
    warn “$pid has gone away!”;
}
```

比发送信号更有趣的可能是捕捉信号。为什么想要这么做呢？假设你有一个程序在*/tmp* 下建立了一些文件，通常，在程序结束时，你需要删除这些文件。如果在程序运行期间，某人按下 Ctrl-C，则在*/tmp* 留下了一些垃圾文件，这是不好的。要

修正这个问题，你可以创建一个 singal handler 来处理清除的问题：

```
my $temp_directory = "/tmp/myprog.$$"    #在这里创建文件
mkdir $temp_directory, 0700 or die "Cannot create $temp_directory: $!";

sub clean_up{
  unlink glob "$temp_directory/*";
  rmdir $temp_directory;
}

sub my_int_handler {
  &clean_up;
  die "Interrrupted, exiting …\n";
}

$SIG{"int"} = 'my_int_handler';

#程序运行一段时间后，在 temp 目录下创建了一些临时文件，可能某人按下了 Ctrl-C
.
#现在是普通运行的结尾
&clear_up;
```

变量%SIG 的赋值将激活 handler,直到撤销为止。key 为没有前缀 SIG 的信号名，值为没有&的子程序名◆。从这里开始，如果有 SIGINT，Perl 会停止当前的处理，跳转到子程序处。我们的子程序清空临时文件并退出。（如果没有按下 Ctrl-C，我们仍在程序的结尾处调用&clean_up。）

◆其值也可以是子程序的引用，但这里我们没有这样做。

如果子程序是返回而不是退出，程序则会继续在它中断的地方执行。这在需要中断而非停止运行的情况下是有用的。例如，假设处理文件的每一行均要花几秒的时间，这是很慢的，你想在中断发生且其不是正在处理一行的时候，将整个进程停止掉。在程序中设置一个 flag，在每一行处理完毕时检查它：

```
my $int_count;
sub my_int_handler {$int_count++}
…
$int_count=0;
while(<SOMEFILE>){
  …某些处理花了几秒钟时间…
  if ($int_count) {
  #中断发生
  print "[processing interrupted…]\n";
  last;
  }
}
```

当每一行处理时，$int_count 的值可能是 0 如果没人按下 Ctrl-C, 因此循环会继续处理下一行。但是，如果中断发生，则中断处理部分会增加$count_int，当在结尾处检测到它时，循环退出。

因此，可以设置 flag 或退出程序，这几乎是捕捉信号最常用的。当前 singal handlers 的实现并非完美无缺的◆，因此，尽量减少这一部分，否则某一天你的程序可能出现你意想不到的事。

◆这是 Perl 开发者要修正的首要问题，因此我们期望在 Perl6 中可靠的信号处理会是其首要的新功能之一。问题是信号可能任何时候发生，甚至在 Perl 还没准备好的时候。如果 Perl 正在分配内存而此时信号产生了，singal handler 可以意外的分配一些内存，而你的程序就死了。当 Perl 代码分配内存时，你是不能控制的，但 XSUB 代码（通常由 C 书写）可以安全的处理信号。参照 Perl 文档，了解更多的关于这些高级话题的信息。

14．8 练习

答案参见附录 A：

1. [6]写一个程序可以转到某个特定的（写入代码中的）目录，如系统的根目录，再执行 ls –l 得到那个目录的目录列表。（如果你的是 non-Unix 系统，使用你自己的系统命令，得到那个目录的详细列表）

2. [10]修改第一题的程序，将结果输出到当前目录的文件 ls.out 中。错误的结果输出到文件 ls.err 中。(你不需要做任何特殊的事，这两个文件中的任意一个都可能是空的。)

3. [8]写一个程序能解析 date 命令的输出，判断当前日期是一个星期的第几天。如果是 weekday（周一至周五），则输出 get to work；否则，输出 go play。date 命令的输出如果是由 Mon 开头，则指星期一◆。如果你的 non-Unix 系统没有 date 命令，则伪造一个程序使之可以输出像 date 命令那样的结果。我们这里给你提供一个两行的程序，如果你答应不问我们它工作的原因：

◆至少当这些星期几是由英文输出的时候。你应当根据你的系统调整它，如果你的系统不是这样。

```
#! /usr/bin/perl
print localtime() . “\n”;
```

第十五章 Perl 模块

Perl 除了本书介绍的之外，还有许多内容，并且还有许多人利用 Perl 做了大量有趣的事。如果有一个问题需要解决，那很可能已经有人解决，并把它放在 CPAN（全面 Perl 归档网络(Comprehensive Perl Archive Network)）上了。它分布在世界各地的服务器及镜像上，其中有成千上万的可以复用的 Perl 代码。

如果想学习如何写模块，可以参见羊驼书(Alpaca book)。本章，我们将介绍如何使用已经存在的模块。

15．1 查找模块

从两种途径可以得到模块：Perl 发布包中附带的以及从 CPAN 中下载并安装的。除非特别说明，我们讨论的模块是 Perl 发布包中附带的。

需要 Perl 发布包中没有的模块，可以在 CAPN 上：http://search.cpan.org，或 kobes' 上：http://kobesearch.cpan.org◆查找。可以在上面找到你需要的模块。

◆是的，URL 中应当是：s's，但没人提到这个问题也没人修正它。

你可以在下载整个包之前先阅读关于此模块的文档。也可以浏览文件，而不用先安装。

在打算网上查找模块之前，先检查其是否已经安装。方法之一是使用 perldoc 相应文档。例如 CGI.pm 是随 Perl 发布的一个模块（我们将在本章后面讨论），你可以如下的阅读其文档：

```
$perldoc CGI
```

尝试阅读一个不存在的模块的文档，将得到一条出错信息：

```
$ perldoc Llamas
$ No documentation found for "Llamas".
```

你系统上的文档，也可能以其它的格式存在，例如 HTML：

◆我们在羊驼书中(Alpaca book)中覆盖了 Perl 的文档，许多模块的文档和实际的代码在同一个文档之中。

15．2 安装模块

当想安装没有的模块时，通常只需下载相应的包，解压，再在 shell 中运行一些命令。*README* 或 *INSTALL* 能给你更多的

信息。如果模块使用了 MakeMaker◆，则需运行的命令大致如下：

◆也就是 Perl 模块：ExtUtils::MakeMaker，随 Perl 一起发布。它能根据已知信息来建立文件，而这个文件包含针对你的系统安装 Perl 所需的指令。

```
$ perl Makerfile.PL
$ make install
```

如果在默认目录中不能安装，你可以针对 *Makerfile.PL* 使用 PREFIX 这个参数来指定目录。

```
$ perl Makerfile.PL    PREFIX=/Users/fred/lib
```

一些模块的创建者使用另一个模块：Module::Build,来构造及安装他们的模块。此时的命令如下：

```
$ perl Build.PL
$./Build install
```

某些模块依赖于其它模块，除非已经把这些模块已经安装上了，否则其不能工作。自己可以不做这些工作，我们可以利用 Perl 自带的一个模块：*CPAN.pm*◆。在命令行中，你可以启用 *CPAN.pm* 这个 shell，其中你可以输入命令。

◆扩展名.pm 表示“Perl Module。”一些流行模块的发音会加上“.pm”来表明其的不同点。例如，CPAN 表示全面 Perl 存档网络时其含义是不同于 CPAN 这个模块的，后者被读作：“CPAN.pm.”

```
$ perl –MCPAN –e shell
```

甚至上述做法也是复杂的，因此以前我们中一位写了一个叫做 cpan 的脚本，它随 Perl 发布。只需在这个脚本后接需要安装的模块的列表。

```
$ cpan Module::CoreList LWP CGI::Prototype
```

你可能说：“我没有命令行！”如果使用的是 ActiveState 移植的 Perl(Windows, Linux，或 Solaris)，你可以使用 Perl 包管理器(Perl Package Manger(PPM))◆，它可以帮助你安装模块。甚至可以使用 ActiveState 的 CD 或 DVD 进行安装◆。你的系统中可能有特有的安装软件的方法，包括 Perl 模块。

◆http://aspn.activestate.com/ASPN/docs/ActivePerl/faq/ActivePerl-faq2.html.

◆你也可以自己制作 CD 或 DVD 来存储资料。虽然 CPAN 现在大约有 3GB，但“minicpan”(也是我们中的一位作者写的)可以将其整理，只保留最新的所有信息，大约为 500MB。参见 CPAN::Mini 模块。

15．3 使用简单的模块

假定在程序中得到了像*/usr/local/bin/perl* 这样的文件名，需要确定其基本名字（basename）。这是很容易的，因为基本名字

是最后一个斜线后的部分（本例中是，*perl*）：

```
my $name = "/usr/local/bin/perl";
(my $basename = $name) =~ s#.*/##;      #Oops!
```

和在前面遇到的一样，Perl 首先进行括号内的赋值运算，再进行替换。替换操作的意思是将任何以斜线结尾的字符串（也就是，路径名部分）用空串替换，基本名字不变。

如果使用这样的代码，看起来能正常工作。是的，它只是看起来能，事实上这里有三个问题。

第一，Unix 文件或目录可以包含换行符。(它虽然不常出现，但是允许的。）由于正则表达式的点(.)不能匹配换行符，因此像"/home/fred/flintstone\n/brontosaurus"这样的字符串就不能正常处理，因为它认为基本名字是"flintstone\n/brontosaurus"。你可以在模式后使用/s 这个选项来解决这个问题（如果你知道这种极少出现的情况），将模式写成：s#.*/##s。第二，这只是针对 Unix 的。这假定了目录的分隔符是正斜线（/）而不是其它系统中的反斜线（\）或冒号。

第三个（最大的）问题是，我们在解决一个别人已经解决过的问题。Perl 自带有许多模块，它们漂亮的扩展了 Perl 的功能。如果这些还不够，你还可以从 CPAN 得到更多的模块，并且每周都有新的模块加入 CPAN。你（或者，更好是，你的系统管理员）在你需要相应的功能时，可以安装它们。

在本节剩余部分，我们将介绍如何使用 Perl 附带的一些模块。(这些模块能作许多事。这里只是如何使用这些模块的一个概括描述。)

我们不能将所有你需要知道的如何使用模块的知识都告诉你，因为使用某些模块需要对一些高级的知识如引用，对象有了解◆。这些知识，包括如何创建模块，在 Alpaca 书中有详细的描述。

◆在下面的讨论中，你可以在不了解这些知识前使用那些使用了引用，对象的模块。

15．3．1． File::Basename 模块

上例中，从文件名（filename）中得到基本名字(basename)的方法不是可移植的。我们说明了某些看起来直观的解决方法是可疑的，某些假定甚至是错误的。(如换行符不会出现在文件和目录名中的假设）我们还重新制造轮子，解决别人已经解决过的问题。

还有一种更好的从文件名得到基本名字的方法。Perl 附带了一个叫做 File::Basename 的模块。使用命令 perldoc File::Basename，或者你系统中的文档，可以阅读其功能，这是使用新模块第一步应做的事情。(但通常是第三步，或第五步)

当你准备好使用它时，在程序顶端用 use 声明它：◆

◆在程序顶端声明模块是一种传统，因为这样有利于维护人员知道你使用的模块。例如，当在一台新机器上安装程序时，将节约你大量的时间。

```
use File::Basename;
```

编译时，Perl 发现这行代码，将相应的模块载入。现在在本程序后面就有了一些新的函数可供使用。我们早期的例子所需的是 basename 函数：

```
my $name = "/usr/local/bin/perl";
my $basename = basename $name;   #得到 'perl'
```

上述代码在 Unix 系统中能正确运行。在 MacPerl, Windows, VMS,或者别的系统中呢？这不会有任何问题，因为这个模块知道你的系统的类型，它会默认使用你的系统的文件名规则。（此时，在$name 存放的 filename 应和你的系统一致。）

这个模块还提供了其它的函数。其中之一是 dirname 函数，它可以从一个全文件名中得到目录名。这个模块还可以让你将文件名及其扩展名分隔开，或者改变默认的文件名规则◆。

◆你可能需要改变文件名的规则，如在一台 Windows 的机器上处理 Unix 机器上的文件名，例如在 FTP 连接上发送命令。

15．3．2．仅使用模块中的一些函数

假定需要在以前的程序中加入 File::Basename，同时你发现有一个子程序叫做：&dirname。因此，你有一个子程序其名字和此模块中的一个函数同名◆。麻烦的地方是这个新的 dirname 已经被作为 Perl 的子程序（在模块内），该怎么办呢？

◆当然，你自己的&dirname 子程序的作用可能也是类似的，这里只是作为一个例子。某些模块有上百个函数，这将提高名字冲突的概率。

例如 File:Basename，在 use 声明中，可以使用输入列表（*import list*）来指定所需的函数，那它将只提供这些函数。下例，将只给出 basename：

```
use File:Basename qw/ basename /;
```

下例，我们不使用任何函数：

```
use File:Baename qw/ /;
```

这通常写成：

```
use File:Basename ( );
```

我们为什么要这么做呢？这明确的要求 Perl 像以前那样载入 File:Basename，但不引入任何的函数名。如果引入的话，我们就可以使用这些短的，简单的函数名如：basename，dirname。如果没有引入这些名字，我们仍然能使用它们。当没有引入时，我们可以使用它们的全名来调用：

```
use File:Basename qw/ /;                    #没有引入函数

my $betty = &dirname($wilma);               #使用我们自己的子程序 &dirname(这里没有显示)

my $name = "/usr/local/bin/perl";
```

```
my $dirname = File::Basename::dirname $name;   #使用模块中的 dirname
```

模块中 dirname 函数的全名是 File::Baename::dirname。我们总是可以使用函数的全名，一旦我们将模块载入，无论是否引入 dirname，我们均可以使用全名。

许多时候，希望使用模块的默认引入列表。但可以使用你自己的列表来改变这种默认行为。使用你自己的列表的另一个理由是，你想引入某些函数，而这些函数不在默认的列表之中，因为许多模块包含（不常用的）函数，这些函数不在默认的引入列表中。

某些模块，在默认的情况下，比别的模块引入更多的记号（symbol）。每一个模块的文档会仔细指出它所引入的记号，当然你总是可以改变这些默认的引入列表，如 File::Basename。引入的就是没有任何记号的空表。

15．3．3．File::Spec 模块

现在你可以找出一个文件的 basename。这是很有用的，但有时你可能希望将它和目录名一起组合成一个全文件名（full filename）。例如，你想在文件名如 */home/rootbeer/ice-2.1.txt* 的 basename 前加入前缀：

```
use File::Basename;

print “Please enter a filename:”;
chomp(my $old_name = <STDIN>);

my $dirname = dirname $old_name;
my $basename = basename $old_name;

$basename =~ s/^/not/;   #在 basename 前加入前缀
my $new_name = “$dirname/$basename”;

rename($old_name, $new_name)
   or warn “Can’t rename ‘old_name’ to ‘$new_name’: $!”;
```

你发现了其中的问题吗？我们又一次假定了文件名是按照 Unix 习惯：在目录名和 basename 之前使用正斜线分开来的。幸运的是，Perl 提供了一个模块来处理这种问题。

模块 File::Spec 用来处理文件规范的(*file specifications*)，如文件名，目录名，以及一些别的存在在文件系统中的信息。同 File::Basename 一样，它知道其操作系统的类型，因此会自动选择正确的规则来处理。File::Spec 和 File::Basename 不同的地方是：它是面向对象(简称“OO”)的模块。

如果对“OO”不了解，不用担心。如果知道对象，那非常好；你可以使用这些“OO”模块。如果不知道对象，也同样无所谓。输入下面的代码，它同样能正常工作。

这里，我们阅读 File::Spec 的文档，想使用其中 catfile 这个方法（*method*）。什么是 *method*？它是一种不同类型的函数，此

时我们只关心这么多。它们的不同点在于，当从 File::Spec 中调用一个方法时，需要使用全名，如：

```
use File::Spec;

#得到上面的$dirname, $basename 的值

my $new_name = File::Spec->catfile($dirname, $basename);

rename($old_name, $new_name)
   or warn “Can’t rename ‘$old_name’ to ‘$new_name’: $!”;
```

方法的全名是模块的名字（此时叫做类），小箭头（->），以及方法的短名字。这里使用小箭头而非像 File::Basename 的双冒号。

由于我们调用方法使用的是全名，那引入了哪些符号呢？一个也没有。这对于 OO 模块是普遍的。你不需要担心你有一个子程序的名字同 File::Spec 中某个方法同名。

觉得使用这些模块很麻烦吗？你如果能保证你的程序只在 Unix 机器上运行，且你完全知道 Unix 的文件名规则，那么你可以将你自己的这些假设写入代码中。但这些模块给你提供了让程序更健壮，更具移植性，并且是不需要付出任何额外代价的方法。

15．3．4．CGI.pm

如果要创建 CGI 程序（我们在本书中不打算全面介绍），可以使用 CGI.pm 模块。不需要处理脚本的接口以及输入解析部分，这些部分让太多人陷入麻烦。CGI.pm 的作者，Lincoln Stein，花了大量时间确保它能在大多数服务器及操作系统中能正确处理。使用这个模块，并把注意力放在脚本中有趣的部分。

CGI 模块提供了两种方法，一种是普通的老式函数接口，一种是 OO 接口。我们使用第一种。和前面一样，我们可以模仿 CGI.pm 模块中的例子。在引入列表(import list)时，我们使用:all，这是一个 *export tag*，将一次指定一组函数而非像在前面模块中看到的那样只是单个函数◆。

◆这个模块还有其它的能选择一组函数 export tags。例如，如果想使用处理 CGI 的那些，可以使用:cgi，如果想使用生成 HTML 的函数，可以使用:html4。查看 CGI.pm 文档了解更多的信息。

```
#! /usr/bin/perl

use CGI qw(:all);

print header(“text/plain”);

foreach my $param ( param())
  {
  print “$param: “. param($param) . ”\n”;
```

```
}
```

如果想以 HTML 的格式输出，CGI.pm 有许多函数可以方便的做到。可以使用 start_html()来处理 HTML 的头，许多的 HTML 标签和相同名字的相应函数，如 h1()指<H1>标签。

```
#! /usr/bin/perl
use CGI qw(:all);

print header(),
    start_html("This is the page title"),
    h1( "Input parameters" );

my $list_items;
foreach my $param ( param() )
        {
         $list_items .= li( "$param: " . param($param) );

print ul ( $list_items );

print end_html();
```

是不是很容易？你不需要知道 CGI.pm 是如何处理的；你只需要相信它能正确进行处理。当把困难的部分交给 CGI.pm 处理时，你就可以集中精力在程序中更有趣的部分。

CGI.pm 还能做更多的事情，如处理 cookies，重定向(redirection)，以及多页窗体（multi-page forms）。你可以从文档的例子中学到更多的知识。

15．3．5.数据库和 DBI

DBI（数据库接口(database interface)）模块不是 Perl 默认附带的，但它是最常用的模块之一，因为大多数用户都需要连接到某种类型的数据库上。DBI 漂亮的地方在于，对于绝大多数常用的数据库，其接口都是一样的，从 csv（comma-separated value）文件到大型的数据库服务器如 Oracle。它有 ODBC 的驱动程序，某些驱动程序是由厂商提供的。想了解全面的详细信息，可以参见 Perl DBI 编程(Porgramming the Perl DBI)（O'Reilly）。也可以查看 DBI 的网站：http://dbi.perl.org/。

当安装了 DBI 后，也需要安装 DBD（数据库驱动程序(database driver)）。从 CPAN 上搜索 DBD，会返回一长串的结果。根据数据库服务器，及其版本安装正确的数据库驱动程序。

DBI 是一个 OO 模块，但不需要完全了解 OO 编程之后才开始使用它；根据文档中的例子就可以开始了。要连接数据库，需要 use DBI 模块，并调用 connect 方法。

```
use DBI;
$dbh = DBI->connect($data_source, $username, $password);
```

$data_source 含有需要使用的 DBD 的信息。对于 PostgreSQL，驱动是 DBD::Pq，则$data_source 看起来如下：

```
my $data_source = “dbi:Pg:dbname=name_of_database”;
```

一旦连接上数据库后，则进入了 preparing, executing, reading 查询的循环。

```
$sth = $dbh->prepare(“SELECT * FROM foo WHERE bla”);
$sth->execute( );
@row_ary = $sth->fetchrow_array;
$sth->finish;
```

当结束时，需要断开和数据库的连接。

```
$dbh->disconnect( );
```

参阅 DBI 的文档了解更多的信息。

15．4 练习

答案参见附录 A：

1. [15]从 CPAN 上安装 Module::CoreList 这个模块。打印出 Perl5.006 所附带的所有模块。创建一个 hash，其 keys 为给定版本的 Perl 所带的模块的名字，使用下面的代码：

   ```
   my %modules = %{%module::CoreList::version{5.006}};
   ```

2. [15]获得当前目录文件名的列表。使用 C<Cwd>模块得到当前的目录，再使用 C<File::Spec>模块将目录名和文件名结合起来得到绝对路径(absolute path)。将此路径输出在标准输出设备上，一行一条。你的解决方案应当可以移植到其它系统之中。

3. [15]利用前一题的输出，将其读入路径的列表中，再使用 C<File::Basename>模块，将文件名从中分离出来。输出文件名。你的解决方案应当可以移植到其它系统之中。

第十六章 一些高级的 Perl 技术

我们到现在为止介绍的都是 Perl 最核心的部分，每一个 Perl 使用者都应当知道。还有些别的技术，它们不是必需的，但也非常有用。我们将其中最重要的部分收集在本章中。

不要被本章的标题所误导；这些技术并不比本书其余部分难于理解。“高级的”含义仅仅指它们对于初学者不是必需的。第一次阅读本书时，你很可能想跳过（或浏览）本章，而直接开始使用 Perl。当你准备了解更多的 Perl 时，再回过头来阅读。把整章看作脚注◆。

◆我们在草稿中是这么打算的，但被 O’Reilly 的编辑坚定的否决了。

16．1 利用 eval 捕获错误

有时，你程序中某段普通代码可能引起严重错误(fatal errors)。例如，下面这些均能让你的程序崩溃：

```
$barney = $fred / $dino;              #除数为 0 的错误?

print “match\n” if /^($wilma)/;     #非法的正则表达式的错误？

open CAVEMAN, fred                   #用户产生的错误？
    or die “Can’t open file ‘$fred’ for input: $!”;
```

如果不怕麻烦，可以捕捉到一些这种类型的错误，但很难考虑周全。（你怎样检测上例中的$wilma 字符串，以保证它是一个合法的正则表达式呢？）庆幸的是，Perl 提供了处理严重错误的方法：将这些代码放入 eval 块中：

```
eval { $barney = $fred / $dino };
```

甚至当$dino 为 0 时，上述代码也不会让程序崩溃。eval 是一个表达式（不是像 while 或 foreach 那样的控制结构），因此结尾处的分号是必须的。

当执行 eval 块时发生了通常的严重错误，eval 块会停止执行，但程序不会崩溃。当 eval 结束时，你想知道它是正常结束的，还是发生了严重错误。这些结果放在特殊变量$@之中。如果 eval 为你捕捉了严重错误，则$@中将有程序失败的原因，可能如：Illegal division by zero at my_program line 12。如果没有错误，则$@为空。这意味着$@是一个有用的 Boolean（true/false）值(真，表示有错误)，因此可能在 eval 块后看到如下的代码：

```
print “An error occurred: $@” if $@;
```

eval 块是真正的一个块，因此其中可以有新的局部（my）变量。下面的程序演示了一个 eval 块：

```
foreach my $person (qw/ fred wilma betty barney dino pebbles /) {
  eval {
    open FILE, "<$person"
      or die "Can't open file '$person': $!";

    my($total, $count);

    while (<FILE>){
      $total += $_;
      $count++;
    }

    my $average = $total/$count;
    print "Average for file $person was $average\n";

    &do_something($person, $average);
  };

  if ($@){
   print "An error occurred ($@), continuing\n";
  }
}
```

这里捕捉了多少严重错误呢？如果打开文件时出了一个错误，这个错误则被捕捉。计算平均数时，除数可能为 0，这个错误也将被捕捉。甚至&do_something 这个子程序，由于其也在 eval 块之内，同样受到 eval 的保护，eval 块会捕捉任何在其运行期间的严重错误。（这个功能是非常方便的，如果调用别人的程序，但你不知道这些代码是否足够强壮，这样能避免让你的程序崩溃。）

当处理某个文件时出现了错误，我们得到这个错误信息，但程序不会因此停下来，会继续处理下一个文件。

可以将 eval 块进行嵌套。内层捕捉它运行时候的错误（当内层的 eval 结束时，你可能希望使用 die 将这些错误在输出来，因此外层的 eval 可以捕捉它。）一个 eval 块捕捉任何它运行时发生的错误，包括其调用的子程序的错误（如前面的例子。）

eval 是一个表达式，这就是为什么需要结尾的分号来关闭花括号的原因。由于是一个表达式，则有一个返回值。如果没有错误，就像任何的子程序一样：返回值为最后一个被求值的表达式的值，或者是由 return 返回的值。下面是另一种进行这种数学运算而不担心除数是否为 0 的做法：

```
my $barney = eval { $fred / $dino };
```

如果 eval 捕捉到了严重错误，则返回值为 undef 或空列表，依赖于其 context。因此，在前面的例子中，$barney 为除法运算的正确值，或者是 undef。我们不需要在继续使用之前检测$@，虽然之前进行检测 defined ($barney)是一个好主意。

有四种类型的问题不能由 eval 捕捉。第一种是严重到可以让 Perl 崩溃的错误，如内存耗光。由于 Perl 没有运行，它不能捕捉这些错误◆。eval 块中的语法错误会在编译时被发现，因此它们也不会出现在$@之中。

◆某些这种类型的错误在 perldiag 的用户手册中被标记有(X)，如果你感兴趣的话。

exit 会立刻终止一个程序的运行，即便是在 eval 块内的子程序中调用的。这明确暗示了，当写子程序时，应当使用 die 而非 exit 来表示程序出了问题。

第四个不能由 eval 块捕捉的问题是 warnings：无论是用户产生的(从 warn 中)或者 Perl 内部产生的（在命令行中使用-w 参数，或者使用 use warnings pragma）。eval 中有一种分离出来的机制来捕捉 warnings。查看 Perl 文档中关于_ _ WARN_ _的论述。

eval 还有一种形式，如果错误使用将非常危险。有时，你可能遇到某人告诉你在代码中使用 eval 可能引起安全问题，因此不要使用它。他们（基本上）是对的，使用 eval 时应当非常小心，但他们说的是另一种形式的 eval，有时被称作"eval of string." 如果关键字后面紧接着的是花括号括起来的代码块，如我们这里的，则没有必要担心 eval 的安全问题。

16．2 使用 grep 在列表得到元素

有时只需要列表中的某些项。例如只需要数字列表中的奇数，或者文本中提到 Fred 的行。如你在本节中将看到的，可以使用 grep 从列表中得到某些项。

让我们来尝试第一个：从一个大的数字列表中得到其中的奇数。这里不需要使用什么新的方法：

```
my @odd_numbers;

foreach (1 .. 1000) {
  push @odd_numbers, $_, if $_ % 2;
}
```

这段代码使用了模运算(%)，在第二章中介绍过。如果数字是偶数，则"模 2"运算的结果是 0，为 false；如果是奇数则得到 1，为 true，因此只有奇数才会被存入数组中。

除了有些长以及执行时间也较长外，上述代码没有任何问题。由于上述原因，Perl 提供了 grep 操作：

```
my @odd_numbers = grep {$_ % 2} 1..1000;
```

上面一行代码即得到 500 个奇数。它是怎样工作的呢？grep 第一个参数是一个块，其中$_依次为列表中的每一个值，返回一个 Boolean(true/false)值。剩下的参数是相应的列表。grep 会首先计算表达式的值，这和 foreach 循环一致。如果块中最后一个表达式的返回值为 true，则这个元素会被返回。

当 grep 运行时，$_会被设为列表中的值，一个接一个。你已经在 foreach 循环中见过这种操作了。通常在 grep 表达式内部修改$_的值是一个坏主意，因为这会破坏原始的数据。

grep 和一个传统的 Unix 工具同名，后者将文件中匹配上正则表达式的行找出来。我们也可以使用 Perl 的 grep，它的功能更强大。下面是我们将出现 fred 的行提取出来：

```
my @matching_lines = grep {/\bfred\b/i} <FILE>;
```

grep 还有一个更简单的用法。如果表达式很简单(而非一个整块)，则可以使用这个表达式，后接逗号(,)，逗号在花括号的位置。下面是这种简单的方法：

```
my @matching_lines = grep /\bfred\b/i, <FILE>;
```

16．3 使用 map 对列表项进行变换

另一个通常的操作是，转变列表项的格式。例如，假设你想将一列数字按照货币数字的格式输出，如第十三章的子程序&big_money。我们不想修改原始数据，只想修改一份拷贝的进行输出。下面是一种方法：

```
my @data = (4.75, 1.5, 2, 1234, 6.9456, 12345678.9, 29.95);
my @formatted_data;

foreach (@data) {
  push @formatted_data, @big_money($_);
}
```

这和本章中介绍 grep 开头处的例子类似，不是吗？替换部分的代码和 grep 这个例子中的代码非常相似：

```
my @data = (4.75, 1.5, 2, 1234, 6.9456, 12345678.9, 29.95);
my @formatted_data = map {&big_money($_) } @data;
```

map 操作和 grep 非常类似，因为它们有相同类型的参数：一个使用$_的块，以及一列需要处理的元素。它们处理的方式也是类似的，首先根据列表中的元素对块的值进行判断，每一次$_被赋予新的列表中的值。但使用块中最后一个表达式的值的方法是不同的：不是返回一个 Boolean 值，而是最终值作为返回的结果◆。任何的 grep 或 map 语句均可以用 foreach 循环重新书写，每一次将结果元素放入一个临时数组中。但短的方式通常更有效以及方便。由于 map 或 grep 的结果是列表，因此它可以直接传递给另一个函数。因此我们可以将格式化后的货币数字打印出来，这些值是被缩进的：

◆另一个重要的不同之处在于，map 所用的表达式是在列表 context 中被求值得，因此可以返回任意数量的元素，而并非一次一个。

```
print “The money numbers are:\n”,
  map { sprintf(“%25s\n”, $_) } @formatted_data;
```

我们可以一次对它进行处理，而不使用临时数组@formatted_data：

```
my @data = (4.75, 1.5, 2, 1234, 6.9456, 12345678.9, 29.95);
print “The money numbers are:\n”,
 map {sprintf(“%25s\n”, &big_money($_))} @data;
```

同在 grep 见到的一样，map 也有更简单的使用法。如果表达式很简单(而非一个整块)，则可以使用这个表达式，后接逗号(,)，逗号在花括号的位置：

```
print “Some powers of two are:\n”,
  map “\t” . (2 ** $_) . “\n”, 0 ..15;
```

16．4 不用双引号的 hash keys

Perl 为程序员提供了许多的简便方法，例如省略掉某些 hash keys 上的双引号。

不能省略掉每一个 key 上的引号，因为 hash key 可以是任意的字符串。但大多时候 keys 是非常简单的。如果 hash key 只由字母，数字，下划线，以及不由数字开头组成，则可以省略掉引号。这些没有引号的简单字符串被叫做 *bareword*。

允许使用这种简写的一个地方，也是 hash key 常出现的地方，是在花括号中引用 hash 元素。例如$score{“fred”}，也可以写做$score{fred}。由于许多的 hash key 是这种类型，不使用引号将非常方便。但注意：如果花括号里面的不仅仅是 *bareword*，Perl 将会把它当作表达式来处理。

另一个 hash keys 常出现的地方是，使用 key/value 对 hash 赋值的时候。大箭头(=>)在 key 和 value 之间是非常有用的因为(如果 key 是 bareword)大箭头为你起了引号的作用：

```
#Hash 中包含了 bowling 的成绩
my %score = (
   barney => 195,
   fred    => 205;
   dino    => 30,
);
```

这是大箭头和逗号一个重要的不同地方；大箭头左边的 bareword 暗含着是由引号括起来的。（而右边的则没有。）大箭头的这个性质不仅仅是针对 hash 的，虽然这是最常使用的地

16．5Slices

通常，我们只需处理列表中的部分元素。例如，Bedrock 图书馆的例子中，我们在一个大文件中保存了其赞助者的信息◆。文件的每一行包含赞助者的由逗号分隔开的 6 项信息：姓名，图书卡号，住址，家庭电话号码，工作住宅电话，当前借出的书数。文件的一部分内容如下：

◆这因该是个功能齐全的数据库，而不仅是一个文件。他们计划在下个冰河世纪后再更新的系统。

```
fred flintsone:2168:301 Cobblestone Way:555-1212:555-2121:3
barney rubble:709918:3128 Granite Blvd:555-3333:555-3438:0
```

这个图书馆的一个应用需要图书卡号，以及借出的书目，不需要其它的数据。可以使用如下代码，来得到这些数据：

```
while (<FILE>) {
```

```
    chomp;
    my @items = split /:/;
    my($card_num, $count) = ($items[1], $items[5]);
      #现在使用这两个变量
  }
```

之后，数组@items 就用不上了，这看起来是一种浪费◆。将结果赋值给一个列表也许更恰当，如下：

◆这不是特别的浪费。上面所出现的技术，是那些不知道 slices 的程序员常用的，将它们写在这里是为了便于比较。

```
my($name, $card_num, $addr, $home, $work, $count) = split /:/;
```

上述方法避免了使用数组@items，但现在又多了四个不需要的标量变量。对于这种情况，一些程序员习惯使用一些虚拟的变量名，如$dummy_1，表明他们并不关心 split 中的相应元素。Larry 认为这也会有很多问题，因此他加入了 undef 的特殊用法。如果将列表的一个元素被赋给 undef，则意味着忽略此元素：

```
my(undef, $card_num, undef, undef, undef, $count) = split /:/;
```

看起来好些了吗？其优点是不需要一些不必要的变量名。其缺点是需要知道有几个 undef，才知道哪一个元素赋给$count。如果列表中的元素很多，这将带来巨大的麻烦。例如，某些人只需要 stat 中的 mtime 值，其代码如下：

```
my(undef, undef, undef, undef, undef, undef, undef,
    undef, undef, $mtime) = stat $some_file;
```

如果使用了错误个数的 undef，则可能得到 atime 或 ctime 等的值，而这很难调试。还有一种更好的方法，Perl 可以对一个列表做索引，就像数组一样，即 *list slice*。由于 mtime 是 stat 返回列表中的第九个元素◆，因此我们可以使用下标得到：

◆它是第十个元素，但由于索引值从 0 开始，因此其索引号为 9。这和数组的情况是一致的。

```
my $mtime = (stat $some_file)[9];
```

列表元素上的括号是必须的（这里是，stat 的返回值）。像下面这样书写是不对的：

```
my $mtime = stat($some_file)[9];      #语法错误!
```

list slice 括号后必须有一个由方括号括起来的的下标表达式。

回到 Bedrock 图书馆的例子，使用的列表是 split 的返回值。我们利用 slice 将元素 1 和元素 5 取出：

```
my $card_num = (split /:/)[1];
my $count = (split /:/)[5];
```

使用像这样的 scalar-context slice（一次得到列表中的一个元素）没什么不可以的，但如果只使用 split 一次将更简单有效。因此让我们一次中完成。在 list context 中使用 list slice，可以一次将所有值取出：

```
my($card_num, $count) = (split /:/)[1,5];
```

上述索引值将元素 1 和元素 5 从列表中取出，将它们按照 2 个元素的列表值返回。它们被赋值给两个 my 变量，其结果正是我们所需的。这里只用了一次 slice，但使用了一个简单表达式给两个变量设值。

slice 通常是将一些元素从列表中取出的最简单方法。下面，我们将列表的第一个元素，最后一个元素取出，你需要知道-1 表示最后一个元素◆：

◆将列表排序，以找到其极值元素，不是最有效率的方法。但 Perl 的排序非常快，因此元素个数在几百以内这种方法也是可以取的。

```
my($first, $last) = (sort @names)[0, -1];
```

slice 的下标可以是任意的序列，甚至可以是重复的。下例将具有十个元素的列表中的五个元素取出：

```
my @names = qw{ zero one two three four five six seven eight nine };
my @numbers = ( @names)[ 9, 0, 2, 1, 0 ];
print "Bedrock @numbers \n";      #输出 Bedrock    nine zero two one zero
```

16．5．1 Array Slice

前面的例子可以更简单一些。当使用 silce 从数组得到元素时（和 list 相应），括号是不需要的，因此可以如下这样：

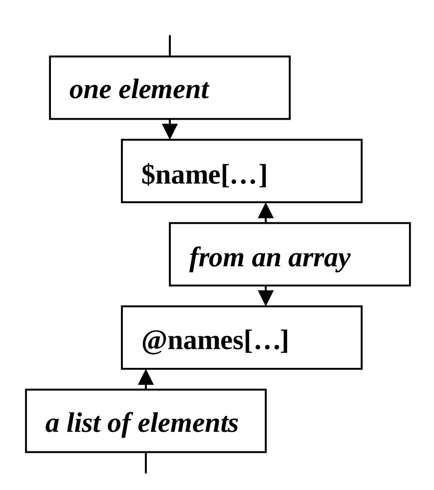
```
my @numbers = @numbers[ 9, 0, 2, 1, 0 ];
```

这不仅仅是省略掉括号的问题；还是一种访问数组元素的不同方法：*array slice*。在第三章中，我们说@names 前面的@标记表示“所有的元素。”实际上，在语言学意义上，这更像一个复数的标记，如“cats”,“dogs”的的“s”符号。在 Perl 中，美元符号($)表示这里是一个元素，但@符号表示这里是一列元素。

slice 总是列表，因此 array slice 使用@标记来表示它。当你在 Perl 程序中见到@names[…]，你需要像 Perl 那样，查看开头处的@符号，以及结尾处的方括号([])。方括号的含义是给数组元素作索引，而@符号表示得到整个列表◆值。美元符号($)表示一个元素。参见图 16-1。

◆当我们说“这个列表时”，并不意味着有多个元素，也可能为空。

图 16-1 数组 slices 和单个元素



变量前的符号（$或@符号）决定了下标表达式的 context。如果前面是$符号，则下标表达式是在标量 context 中被求值，得到一个索引值。如果之前为@符号，则下标表达式在列表 context 中被求值，得到一列索引值。

因此@name[2, 5]的含义同($names[2], $name[5])一样。如果想得到这个列表的值，可以使用 array slice。任何你想用列表的地方，均可以使用 array slice。

有个地方只有 slice 可以使用，但 list 不行：slice 可以在数组中以内插值被替换：

```
my @names = qw{ zero one two three four five six seven eight nine };
print "Bedrock @names[9, 0, 2, 1, 0]\n";
```

如果对@names 内插，将得到数组中的所有元素，由空格分开。如果对@names[9, 0, 2, 1, 0] 内插，它只得到数组中的相应元素，也由空格分开◆。让我们再回到 Bedrock 图书馆的例子中，也许我们的程序需要更新赞助者文件中 Mr.Slate 的地址以及电话号码，因为它搬到了 Hollyrock 山的一个更大住处去了。如果在@items 中有他的信息，我们可以像下面这样做，来更新这两个元素的信息：

◆更准确的说，列表的这些元素被 Perl 的$"变量的值隔开，其默认值为空格。不因该改变其值。当内插值时，Perl 内部会作 join $", @list，其中@list 表示列表表达式。

```
my $new_home_phone = "555-6099";
my $new_address = "99380 Red Rock West";
@items[2, 3] = ($new_address, $new_phone_phone);
```

再一次提醒，使用 array slice 使代码看起来更紧凑。在上例中，最后一行也可以写做（$items[2], $items[3]），但前者更紧凑和有效。

16．5．2Hash Slice

除了 *array slice*，我们也有 *hash slice*。可以对 hash 使用 slice。还记得三人去打保龄球的例子中，我们将他们的成绩记录在

%score 这个 hash 中吗？我们既可以使用 hash 元素的列表来得到他们的成绩，也可以使用 slice。这两种技术是等价的，虽然第二种更简单效率：

```
my @three_score = ($score{"barney"}, $score{"fred"}, $score{"dino"});
my @three_scores = @score{ qw/ barney fred dino/ };
```

slice 总是列表，因此 hash slice 总是使用@符号来表示◆。当在 Perl 程序中见到@score{ … }，你需要像 Perl 那样，查看开头处的@符号，以及结尾处的花括号({})。方括号的含义是给 hash 的元素作索引，而@符号表示得到整个列表◆值。美元符号($)表示一个元素。参见图 16-2。

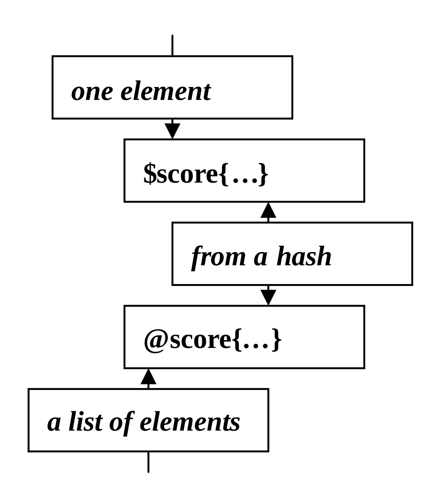
◆这看起来像重复，我们只是想强调 hash slices 和 array slices 是类似的。如果你不这样认为，但我们确实是想强调 hash slices 和 array slices 是类似的。

同 array slice 一样，变量前的符号（$或@符号）决定了下标表达式的 context。如果前面是$符号，则下标表达式是在标量 context 中被求值，得到一个值◆。如果之前为@符号，则下标表达式在列表 context 中被求值，得到一列值。

◆这里有一个例外，但很难遇到，因为在当今的 Perl 代码中，很少使用它。参见 perlvar 的用户手册关于$; 的部分。

当我们谈论 hash 时，猜想为什么没有百分号(%)是很正常的。这个标记表明整个 hash；hash slice(如别的 slice)总是 *list* 而非 hash◆。在 Perl 中，美元符号($)指单个元素，@符号表示有一列值，而百分号表示整个 hash。

图 16-2 Hash slices 和单个元素



◆hash slice 是一个 slice(而非 hash)，如同 house fire 是指 fire（而非 house），而 fire house 是指 house(而非 fire)。大致如此。

如在 array slices 中见到的；hash slice 可以替代相应的 hash 列表元素。因此，可以按如下的方法给我们的朋友的保龄球计分（不需要 hash 中的别的元素）：

```
my @players = qw/ barney fred dino /;
my @bowling_scores = (195, 205, 30);
@score{ @players } = @bowling_scores;
```

最后一行和($score{“barney”}, $score{“fred”}, $score{“dino’})的作用是一样的。

hash slice 也是可以被内插。下面，我们打印出我们最喜欢的保龄球手的成绩：

```
print “Tonight’s players were: @players\n”;
print “There score were: @score{players}\n”;
```

16．6 练习

答案参见附录 A：

1．[30]写一个程序，从文件中读入字符串，一行一个字符串，然后让用户输入模式，这个模式可能匹配上某些字符串。对于每一个模式，程序将指出文件中有多少个字符串（多少行）匹配上了，并指出是哪些。对于新的模式不需要重新读文件，将这些字符串保留在内存中。文件名可以直接写在程序之中。如果模式无效（例如，圆括号不匹配），则程序报告这个错误，并让用户继续尝试新的模式。当用户输入一个空行，则程序退出。

练习题答案

答案略，如果需要请不要向译者索取☺。本书英文版有答案，这个建议是免费的（译者注）。